A L G O R I T M O S

em linguagem C

Paulo Feofiloff

A L G O R I T M O S
em linguagem C

Copidesque: Caravelas Produções Editoriais

Editoração eletrônica: Paulo Feofiloff

Revisão gráfica: Marília Pinto de Oliveira e Wilton Palha

Elsevier Editora Ltda. Conhecimento sem fronteiras
Rua Sete de Setembro, 111, 16º andar
20050-006 – Rio de Janeiro, RJ – Brasil

Rua Quintana, 753, 8º andar
04569-011 – Brooklin – São Paulo, SP – Brasil

Serviço de Atendimento ao Cliente
0800-265340
sac@elsevier.com.br

ISBN: 978-85-352-3249-3

Nota: Muito zelo e técnica foram empregados na edição desta obra. Apesar disso, podem ter ocorrido erros de digitação, de impressão ou de conceitos. Em qualquer das hipóteses, solicitamos a comunicação à nossa Central de Atendimento, para que possamos esclarecer ou encaminhar a questão.

Nem a editora nem o autor assumem qualquer responsabilidade por eventuais danos ou perdas originados do uso desta publicação.

CIP-BRASIL. CATALOGAÇÃO-NA-FONTE
SINDICATO NACIONAL DOS EDITORES DE LIVROS, RJ

F383a

Feofiloff, Paulo, 1946-
Algoritmos / Paulo Feofiloff. – Rio de Janeiro : Elsevier, 2009.
il.

Inclui bibliografia e índice
ISBN 978-85-352-3249-3

1. Algoritmos. I. Título.

08-3451. CDD: 518.1
CDU: 519.165

14.08.08 18.08.08 008208

2ª impressão, corrigida

Prefácio

"Computer science is no more about computers
than astronomy is about telescopes."
— E. W. Dijkstra

"A good algorithm is like a sharp knife:
it does what it is supposed to do with a minimum amount of applied effort."
— T. Cormen, C. Leiserson, R. Rivest, C. Stein, *Introduction to Algorithms*

"A Computação anda sobre três pernas: a correção, a eficiência e a elegância."
— I. Simon

Este livro contém um curso introdutório de algoritmos e estruturas de dados. Ele é apropriado para estudantes de computação que já tenham alguma experiência de programação em linguagem C.[1] (Alguns conceitos e recursos da linguagem estão resumidos nos apêndices.)

O livro discute algoritmos clássicos para vários problemas computacionais básicos. O estudo cuidadoso dessas soluções clássicas deve ajudar o leitor a criar algoritmos para os seus próprios problemas. O texto evita oferecer "receitas" que possam ser aplicadas mecanicamente, e portanto está mais próximo do "ensinar a pescar" que do "dar o peixe".

A coleção de tópicos não difere muito da que se encontra em outras obras sobre o assunto. Mas a abordagem tem as seguintes peculiaridades:

- o destaque dado aos algoritmos recursivos;
- o uso de invariantes na análise de algoritmos iterativos;
- a atenção dispensada à documentação;
- o cuidado com a elegância do código.

O texto evita longas explicações informais sobre o funcionamento e a lógica dos algoritmos, preferindo colocar o leitor em contato direto com o código. Para

[1] Veja o verbete *C programming language* na Wikipedia [21].

ajudar o leitor a entender o código, oferece uma boa documentação e exibe os invariantes dos processos iterativos.

O livro procura mostrar vários algoritmos para um mesmo problema e várias maneiras de escrever o código de um mesmo algoritmo. Além disso, dá exemplos de erros comumente cometidos por programadores inexperientes. Assim, tenta habituar o leitor a distinguir o bom código do mau.

Algoritmos corretos, eficientes e elegantes. Todo bom algoritmo e programa[2] tem três qualidades fundamentais: correção,[3] eficiência[4] e elegância.[5] O livro discute essas três qualidades, de maneira muito informal, através de exemplos.

Antes de aprender a construir algoritmos corretos, é preciso aprender a verificar se um algoritmo dado está correto. A verificação da correção de um algoritmo é uma atividade semelhante à prova de um teorema. (Mas o leitor não deve ficar assustado: o livro trata essa analogia de maneira muito informal.) A verificação depende do enunciado preciso *do que* o algoritmo deve fazer; esse enunciado constitui a *documentação* do algoritmo. No caso de um algoritmo iterativo, a prova da correção se apoia sobre o conceito de *invariante*.

Para provar a correção de um algoritmo, o livro não recorre a abstrações vagas mas estuda a relação entre os valores das variáveis em pontos estratégicos do código. A propósito, convém ouvir E. W. Dijkstra [6]:

> "[We must] deal with all computations possible under control of a given program by ignoring them and working with the program. We must learn to work with program *texts* while (temporarily) ignoring that they admit the interpretation of executable code."[6]

---

[2] Um programa é uma implementação concreta de um algoritmo. Para simplificar a linguagem, o livro tende a usar as duas palavras como sinônimas.

[3] Um algoritmo é **correto** se faz o que dele se espera, ou seja, se cumpre o que sua documentação promete.

[4] Um algoritmo é **eficiente** se não desperdiça tempo. (Uma definição mais ampla diria também que não desperdiça espaço de memória.) Dados dois algoritmos para um mesmo problema, o primeiro é mais eficiente que o segundo se a execução do primeiro consome menos tempo que a do segundo.

[5] Um algoritmo é **elegante** se for simples, limpo, bonito, sem enfeites. Um algoritmo elegante não trata casos especiais do problema em separado. Um algoritmo elegante não tem código supérfluo, nem variáveis desnecessárias, nem construções convoluídas e "espertas", nem sutilezas evitáveis.

[6] "Para lidar com todas as computações possíveis sob o controle de um dado programa é preciso ignorar essas computações e trabalhar com o [texto do] programa. Precisamos aprender a trabalhar com os *textos* dos programas e esquecer (temporariamente) que eles podem ser interpretados como código executável."

E ainda:

> "[...] all by itself, a program is no more than half a conjecture. The other half of the conjecture is the *functional specification* the program is supposed to satisfy. The programmer's task is to present such complete conjectures as proven theorems."[7]

(Espero que o leitor não me considere demasiado pretensioso por citar Dijkstra. O livro procura incorporar um pouco do espírito das observações, mas não pretende implementá-las de maneira séria e sistemática.)

A eficiência dos algoritmos — ou melhor, o consumo de tempo em função do tamanho das instâncias — é analisada de maneira informal e intuitiva, como convém a um livro introdutório.

Além de correto e eficiente, um bom algoritmo deve ser elegante. O conceito é um tanto subjetivo, mas programadores experientes concordam entre si, em geral, quando julgam a elegância de um algoritmo. Para dar uma pálida ideia do conceito, o Capítulo 3 exibe pequenos exemplos de algoritmos elegantes e deselegantes para um mesmo problema.

A estrutura do livro. O livro tem quinze capítulos e um grande número de apêndices. Os três primeiros capítulos constituem uma espécie de introdução. Os doze capítulos seguintes tratam do assunto central do livro. Os apêndices fazem um resumo dos principais conceitos e recursos da linguagem C.

Sugiro começar a leitura pelo Apêndice A, que trata do leiaute de programas. Esse assunto é mais importante do que parece, porque programas precisam ser lidos e compreendidos por seres humanos. Em seguida, sugiro ler os três capítulos introdutórios:

Capítulo 1 (Documentação e invariantes). Apresenta, de maneira muito informal, a ideia da boa documentação e o conceito de invariante de um processo iterativo.

Capítulo 2 (Algoritmos recursivos). O conceito de recursão é fundamental em computação, embora nem sempre receba a atenção que merece. O capítulo introduz a noção de algoritmo recursivo por meio de um exemplo muito simples.

Capítulo 3 (Vetores). Os problemas de busca, remoção e inserção em um vetor são usados aqui como pretexto para ilustrar os conceitos de correção, eficiência e elegância de algoritmos e de código. Em particular,

[7] "Por si só, um programa é apenas metade de uma conjectura. A outra metade da conjectura é a especificação funcional que o programa deve satisfazer. É tarefa do programador apresentar as duas metades de tais conjecturas como teoremas demonstrados."

o capítulo procura despertar a sensibilidade do leitor para o conceito de elegância. Os três problemas — busca, inserção e remoção — reaparecem, em outros contextos, nos capítulos seguintes.

Depois desta introdução, o leitor pode passar ao assunto central do livro:

Capítulos 4, 5 e 6 (Listas encadeadas, Filas e Pilhas). Estes capítulos tratam da manipulação de três estruturas de dados muito úteis, comuns a um sem-número de aplicações.

Capítulo 7 (Busca em vetor ordenado). O capítulo discute o célebre algoritmo da busca binária. As ideias subjacentes ao algoritmo reaparecem em vários dos capítulos seguintes.

Capítulos 8, 9, 10 e 11 (Algoritmos de ordenação). Este grupo de capítulos trata do clássico problema de colocar um vetor de números em ordem crescente. O Capítulo 8 discute dois algoritmos muito simples, mas pouco eficientes. Os demais introduzem algoritmos mais sofisticados e bem mais eficientes. As estruturas de dados criadas por esses algoritmos são muito úteis em outras aplicações, diferentes da ordenação.

Capítulo 12 (Algoritmos de enumeração). Consideramos aqui o problema de gerar todos os subconjuntos de um conjunto. A solução deste problema é uma boa demonstração do poder da recursão. Algoritmos do tipo discutido aqui aparecem em aplicações que envolvem *backtracking*.

Capítulo 13 (Busca de palavras em um texto). O problema de encontrar uma ocorrência de uma dada palavra num texto é um componente básico de muitas aplicações práticas, como a construção de editores de texto e a procura por um gene num genoma.

Capítulos 14 e 15 (Árvores binárias e Árvores de busca). O primeiro destes capítulos introduz uma estrutura de dados fundamental muito útil. O segundo capítulo usa a estrutura para generalizar a busca binária discutida no Capítulo 7.

Exercícios. Os exercícios são parte essencial do livro. Eles esclarecem pontos obscuros e levam o leitor a pensar sobre os detalhes dos algoritmos discutidos no texto. Alguns convidam o leitor a investigar, por conta própria, assuntos que o texto não aborda. A solução de alguns exercícios é dada no Apêndice L.

Um bom número de exercícios explora maneiras alternativas de codificar os algoritmos discutidos no texto. Esses exercícios incentivam o leitor a analisar código e a encontrar defeitos.

Recomendo que o leitor não se limite aos exercícios do livro e aventure-se a resolver os problemas de competições de programação, como o *Programming Challenges* [19] e o *Problem Set Archive* [14]. Os dois livros de Bentley [2, 1] também são excelente fonte de exercícios, exemplos e inspiração.

Histórico. O livro evoluiu a partir das notas de aula que mantenho na Internet (`www.ime.usp.br/~pf/algoritmos/`) há vários anos. Aquelas notas, por sua vez, foram escritas ao longo de várias edições da disciplina Princípios de Desenvolvimento de Algoritmos do curso de graduação em Ciência da Computação da USP (Universidade de São Paulo), administrado pelo IME (Instituto de Matemática e Estatística). Esta disciplina é oferecida no segundo semestre do currículo, logo depois de uma disciplina de introdução à programação.

Agradecimentos. Quero registrar minha gratidão aos alunos, colegas e professores que contribuíram com ideias e material, e corrigiram muitos dos meus erros (não raro fundamentais). Ainda que muitos deles tenham sido corrigidos, é quase certo que muitos outros escaparam.

Agradeço ao Departamento de Ciência da Computação do IME-USP pelo uso de suas instalações e equipamento durante a preparação do livro.

São Paulo, agosto de 2008
P.F.

Sumário

Capítulo 1

Documentação e invariantes

"Let us change our traditional attitude to the construction of programs.
Instead of imagining that our main task is to instruct a computer what to do,
let us imagine that our main task is to explain to human beings
what we want a computer to do."

"Programming is best regarded as the process of creating works of literature,
which are meant to be read."

— D. E. Knuth, *Literate Programming*

"... a program is no more than half a conjecture.
The other half of the conjecture is the functional specification
the program is supposed to satisfy."

— E. W. Dijkstra, manuscrito EWD1036

Há quem diga que *documentar* um programa é o mesmo que escrever muitos comentários de mistura com o código. Essa ideia está errada. Uma boa documentação evita sujar o código com comentários e limita-se a explicar

o que cada uma das funções que compõem o programa faz.

A documentação de uma função é um minimanual que dá instruções precisas e completas sobre o uso correto da função. O minimanual começa por especificar o que "entra" — que dados a função recebe — e o que "sai" — que objetos a função devolve. Em seguida, descreve a relação entre o que entra e o que sai (bem como as eventuais transformações executadas sobre o que entrou). Com isso, uma boa documentação coloca nas mãos do leitor as condições necessárias para detectar os erros que o autor da função tenha porventura cometido.

Em geral, uma boa documentação não perde tempo tentando explicar *como* uma função faz o que faz (o leitor interessado nesta questão deve ler o código). A distinção entre *o que* uma função faz e *como* ela faz o que faz é essencial

para uma boa documentação. Esta distinção é a mesma que existe entre a interface (arquivo `.h`) e a implementação (arquivo `.c`) de uma biblioteca em linguagem C. A seguinte analogia pode tornar a distinção mais clara: Uma empresa de entregas promete apanhar o seu pacote em São Paulo e entregá-lo no dia seguinte no Rio de Janeiro. Isto é *o que* a empresa faz. *Como* o serviço é feito — se o transporte é terrestre, aéreo ou marítimo, por exemplo — é assunto interno da empresa.

1.1 Exemplo de documentação

A função abaixo calcula o valor de um elemento máximo de um vetor. Observe como a documentação da função é simples e precisa:

```
/* A função abaixo recebe um inteiro n >= 1 e um vetor v e
 * devolve o valor de um elemento máximo de v[0..n-1]. */ [1]
int Max (int v[], int n) {
   int j, x = v[0];
   for (j = 1; j < n; j++)
      if (x < v[j]) x = v[j];
   return x;
}
```

A documentação diz o que a função faz mas não perde tempo tentando explicar como a função faz o que faz (se é recursiva ou iterativa, se percorre o vetor da esquerda para a direita ou vice-versa etc.). Para fazer contraste com este bom exemplo, seguem algumas amostras de má documentação:

1. "`a função devolve o valor de um elemento máximo de um vetor`" é indecentemente vago;
2. "`a função devolve o valor de um elemento máximo do vetor v`" ainda é muito vago, pois não explica o papel do parâmetro `n`;
3. "`a função devolve o valor de um elemento máximo de um vetor v que tem n elementos`" é melhor, mas ainda está vago: não se sabe se o vetor é `v[0..n-1]` ou `v[1..n]`;
4. "`a função devolve o valor de um elemento máximo de v[0..n-1]`" já está quase bom, mas sonega a importante restrição "`n >= 1`".

Observe que a documentação menciona *todos* os parâmetros da função (a saber, `v` e `n`) e não faz menção de quaisquer outras variáveis. Observe também a

[1] A expressão `v[`i`..`m`]` representa um vetor `v` indexado por $i, i+1, \ldots, m$.

ausência de comentários inúteis (como, por exemplo, "`o índice j vai percorrer o vetor`" e "`x armazena o maior valor encontrado até agora`") misturados ao código.

Exercícios

1.1.1 Escreva a documentação correta da função abaixo.

```
int soma (int n, int v[]) {
   int i, x = 0;
   for (i = 0; i < n; i++) x += v[i];
   return x; }
```

1.1.2 Escreva a documentação correta da função abaixo.

```
int onde (int x, int v[], int n) {
   int j = 0;
   while (j < n && v[j] != x) j += 1;
   return j; }
```

1.1.3 Critique a seguinte documentação de uma função: "Esta função recebe números inteiros `p`, `q`, `r`, `s` e devolve a média aritmética de `p`, `q`, `r`."

1.1.4 Critique a seguinte documentação de uma função: "Esta função recebe números inteiros `p`, `q`, `r` tais que `p <= q <= r` e devolve a média aritmética dos três números."

1.1.5 Leia o verbete *Software documentation* na Wikipedia [21].

1.2 Invariantes

O corpo de muitas funções contém um ou mais processos iterativos (tipicamente controlados por um `for` ou um `while`). O programador pode enriquecer a documentação da função dizendo quais os invariantes dos processos iterativos. Um **invariante** é uma relação entre os valores das variáveis que *vale no início de cada iteração* do processo iterativo. Os invariantes explicam o funcionamento do processo iterativo e permitem provar, por indução, que ele tem o efeito desejado.

Considere, por exemplo, a função `Max` da Seção 1.1. O processo iterativo controlado pelo `for` tem o seguinte invariante: *no início de cada iteração* (imediatamente antes da comparação de `j` com `n`),

`x` *é um elemento máximo de* `v[0..j-1]`.

O invariante vale, em particular, no início da última iteração, quando `j` vale `n`. Isto mostra que a função de fato devolve o valor de um elemento máximo de `v[0..n-1]`.

```
int Max (int v[], int n) {
   int j, x;
   x = v[0];
   for (j = 1; j < n; j++)
      /* x é um elemento máximo de v[0..j-1] */
      if (x < v[j])  x = v[j];
   return x;
}
```

(O enunciado de um invariante é, provavelmente, o único tipo de comentário que vale a pena inserir no corpo de uma função.)

Exercícios

1.2.1 Mostre que o invariante da função `Max` vale no início da primeira iteração. Suponha que o invariante vale no início de uma iteração qualquer e mostre que ele vale no início da iteração seguinte. Suponha que o invariante vale no início da última iteração e deduza daí que a função devolve um elemento máximo do vetor `v[0..n-1]`.

1.2.2 Qual o invariante do processo iterativo da função `soma` no Exercício 1.1.1?

1.2.3 Qual o invariante do processo iterativo da função `onde` no Exercício 1.1.2?

1.2.4 PROJETO DE PROGRAMAÇÃO. O **piso** de um número x é o único inteiro i tal que $i \leq x < i+1$. O piso de x é denotado por $\lfloor x \rfloor$. Escreva uma função `lg` que receba um inteiro positivo n e calcule $\lfloor \log_2 n \rfloor$. Segue uma amostra de valores:

n	15	16	31	32	63	64	127	128	255	256	511	512
$\lfloor \log_2 n \rfloor$	3	4	4	5	5	6	6	7	7	8	8	9

Documente sua função e enuncie os invariantes do processo iterativo.

Implemente um programa completo que use a função `lg` para imprimir uma tabela de valores de $\lfloor \log_2 n \rfloor$. Faça um bom leiaute do seu programa (veja Apêndice A) e documente-o corretamente.

Durante a fase de testes, o seu programa deve verificar a correção da função `lg` valendo-se da função `log` que faz parte da bibilioteca `math` (veja Seção K.3).

Capítulo 2

Recursão

"Ao tentar resolver o problema, encontrei obstáculos dentro de obstáculos.
Por isso, adotei uma solução recursiva."
— um aluno

"To understand recursion, we must first understand recursion."
— folclore

O conceito de recursão é de fundamental importância em computação. Este capítulo introduz o conceito por meio de um exemplo muito simples.

2.1 Algoritmos recursivos

Muitos problemas computacionais têm a seguinte propriedade: cada instância[1] do problema contém uma instância menor do mesmo problema. Dizemos que esses problemas têm *estrutura recursiva*. Para resolver um tal problema é natural aplicar o seguinte método:

se a instância em questão é pequena,
    resolva-a diretamente (use força bruta se necessário);
senão,
    reduza-a a uma instância menor do mesmo problema,
    aplique o método à instância menor
    e volte à instância original.

A aplicação deste método produz um *algoritmo recursivo*.

[1] Uma **instância** de um problema é um exemplo do problema. Cada conjunto de dados de um problema define uma instância. (A palavra *instância* é um neologismo importado do inglês. Ela está sendo empregada aqui no sentido de *exemplo*, *espécime*, *amostra*.)

2.2 Um exemplo: o problema do máximo

Considere o problema de determinar *o valor de um*[2] *elemento máximo de um vetor* $v[0..n-1]$. O tamanho de uma instância do problema é n. É claro que o problema só faz sentido se o vetor não for vazio, ou seja, se $n \geq 1$. Se $n = 1$,[3] então $v[0]$ é o único elemento do nosso vetor e portanto $v[0]$ é o máximo. Se $n > 1$, o valor que procuramos é o maior dentre o máximo do vetor $v[0..n-2]$ e o número $v[n-1]$. Assim, a instância $v[0..n-1]$ do problema fica reduzida à instância $v[0..n-2]$. Estas observações levam à seguinte função recursiva:

```
/* Ao receber v e n >= 1, esta função devolve o valor de
 * um elemento máximo do vetor v[0..n-1]. */
int MáximoR (int v[], int n) { [4]
   if (n == 1)
      return v[0];
   else {
      int x;
      x = MáximoR (v, n - 1);
      if (x > v[n-1])
         return x;
      else
         return v[n-1];
   }
}
```

Para verificar que uma função recursiva está correta, use o seguinte roteiro. Passo 1: Escreva *o que* a função deve fazer (veja Capítulo 1). Passo 2: Verifique se a função de fato faz o que deveria quando n é pequeno ($n = 1$, no nosso exemplo). Passo 3: Imagine que n é grande ($n > 1$, no nosso exemplo) e suponha que a função fará a coisa certa se no lugar de n tivermos algo menor que n. Sob esta hipótese, verifique que a função faz o que dela se espera.

Como o computador executa uma função recursiva? Embora relevante, esta pergunta será ignorada por enquanto. Veja o conceito de pilha de execução na Seção 6.5.

[2] Eu não disse "*do* elemento máximo" porque o vetor pode ter vários elementos máximos.

[3] Embora sejam tipograficamente semelhantes, os sinais $=$ e `=` têm significados diferentes. O primeiro é o sinal de igualdade da matemática: "$x = y$" significa "x é igual a y". O segundo é o operador de atribuição na linguagem C: "`x = y`" significa "atribua à variável `x` o valor da variável `y`". O "=" da matemática corresponde ao "==" da linguagem C.

[4] Veja Seção A.4.

Exercícios

2.2.1 Escreva uma versão iterativa da função `MáximoR`.

2.2.2 Critique a função abaixo. Ela promete encontrar o valor de um elemento máximo de `v[0..n-1]`.

```
int máximoR1 (int v[], int n) {
   int x;
   if (n == 1) return v[0];
   if (n == 2) {
      if (v[0] < v[1]) return v[1];
      else return v[0]; }
   x = máximoR1 (v, n - 1);
   if (x < v[n-1]) return v[n-1];
   else return x; }
```

2.2.3 Critique a seguinte função recursiva que promete encontrar o valor de um elemento máximo do vetor `v[0..n-1]`.

```
int máximoR2 (int v[], int n) {
   if (n == 1) return v[0];
   if (máximoR2 (v, n - 1) < v[n-1])
      return v[n-1];
   else
      return máximoR2 (v, n - 1); }
```

2.2.4 Se `X` é a função recursiva abaixo, qual o valor de `X(4)`?

```
int X (int n) {
   if (n == 1 || n == 2) return n;
   else return X (n - 1) + n * X (n - 2); }
```

2.2.5 O que há de errado com a seguinte função recursiva?

```
int XX (int n) {
   if (n == 0) return 0;
   else return XX (n/3 + 1) + n; }
```

2.2.6 Programa de teste. Escreva um pequeno programa para testar a função recursiva `MáximoR`. O seu programa deve pedir ao usuário que digite uma sequência de números ou gerar um vetor aleatório (veja Apêndice I).

Importante: Para efeito de testes, acrescente ao seu programa uma função auxiliar que *confira* a resposta produzida por `MáximoR`.

2.3 Outra solução recursiva do problema

A função `MáximoR` discutida acima aplica a recursão ao subvetor $v[0\,..\,n{-}2]$. É possível escrever uma versão que aplique a recursão ao subvetor $v[1\,..\,n{-}1]$:

```
/* Ao receber v e n >= 1, esta função devolve o valor de
 * um elemento máximo do vetor v[0..n-1]. */
int Máximo (int v[], int n) {
   return MaxR (v, 0, n);
}

/* Esta função recebe v, i e n tais que i < n e devolve
 * o valor de um elemento máximo do vetor v[i..n-1]. */
int MaxR (int v[], int i, int n) {
   if (i == n-1) return v[i];
   else {
      int x;
      x = MaxR (v, i + 1, n);
      if (x > v[i]) return x;
      else return v[i];
   }
}
```

A função `Máximo` é apenas uma "embalagem"; o serviço pesado é executado pela função recursiva `MaxR`, que resolve um problema mais geral, com mais parâmetros que o original.

A necessidade de generalizar o problema ocorre com frequência na construção de algoritmos recursivos. O papel dos novos parâmetros (como `i` no exemplo acima) deve ser devidamente explicado na documentação da função,[5] o que nem sempre é fácil (veja mais exemplos nas Seções 7.7 e 12.3).

Exercícios

2.3.1 Verifique que a seguinte função é equivalente à função `Máximo`. Ela usa a aritmética de endereços mencionada no Seção D.4.

```
int máximo2r (int v[], int n) {
   int x;
   if (n == 1) return v[0];
   x = máximo2r (v + 1, n - 1);
   if (x > v[0]) return x;
   return v[0]; }
```

2.3.2 MAX-MIN. Escreva uma função recursiva que calcule a diferença entre o valor de um elemento máximo e o valor de um elemento mínimo do vetor `v[0..n-1]`.

[5] Explicações do tipo "a primeira chamada da função deve ser feita com `i` = 0" não explicam nada e devem ser evitadas a todo o custo.

2.3.3 SOMA. Escreva uma função recursiva que calcule a soma dos elementos positivos do vetor de inteiros $v[0\,..\,n-1]$. O problema faz sentido quando $n = 0$? Quanto deve valer a soma neste caso?

2.3.4 SOMA DE DÍGITOS. Escreva uma função recursiva que calcule a soma dos dígitos decimais de um inteiro positivo. A soma dos dígitos de 132, por exemplo, é 6.

2.3.5 PISO DE LOGARITMO. Escreva uma função recursiva que calcule $\lfloor \log_2 n \rfloor$, ou seja, o piso do logaritmo de n na base 2. (Veja Exercício 1.2.4.)

2.3.6 FIBONACCI. A sequência de Fibonacci é definida assim: $F_0 = 0$, $F_1 = 1$ e $F_n = F_{n-1} + F_{n-2}$ para $n > 1$. Escreva uma função recursiva que receba n e devolva F_n. Escreva uma versão iterativa da função. Sua função recursiva é tão eficiente quanto a iterativa? Por quê?

2.3.7 Seja `F` a versão recursiva da função de Fibonacci (veja Exercício 2.3.6). O cálculo de `F(3)` provoca a sequência de invocações da função dada abaixo (note a indentação). Dê a sequência de invocações da função provocada pelo cálculo de `F(5)`.

```
F(3)
  F(2)
    F(1)
    F(0)
  F(1)
```

2.3.8 Execute a função `ff` abaixo com argumentos `7` e `0`.

```
int ff (int n, int ind) {
   int i;
   for (i = 0; i < ind; i++)
      printf ("  ");
   printf ("ff (%d,%d)\n", n, ind);
   if (n = 1)
      return 1;
   if (n % 2 == 0)
      return ff (n/2, ind + 1);
   return ff ((n-1)/2, ind + 1) + ff ((n+1)/2, ind + 1); }
```

2.3.9 EUCLIDES. A seguinte função calcula o maior divisor comum dos inteiros positivos `m` e `n`. Escreva uma função recursiva equivalente.

```
int Euclides (int m, int n) {
   int r;
   do {
      r = m % n;
      m = n; n = r;
   } while (r != 0);
   return m; }
```

2.3.10 EXPONENCIAÇÃO. Escreva uma função recursiva eficiente que receba inteiros

positivos k e n e calcule o valor de k^n. Suponha que k^n cabe em um `int` (veja Seção C.2). Quantas multiplicações sua função executa aproximadamente?

2.3.11 Leia o verbete *Recursion* na Wikipedia [21].

Capítulo 3

Vetores

Um **vetor** é uma estrutura de dados que armazena uma sequência[1] de objetos, todos do mesmo tipo, em posições consecutivas da memória (veja Seção D.4). Este capítulo estuda os problemas de procurar um objeto em um vetor, de inserir um novo objeto no vetor e de remover um elemento do vetor. Os problemas servem de pretexto para ilustrar os conceitos de correção, eficiência e elegância de algoritmos (veja notas de rodapé na página vi do prefácio), bem como para exibir alguns exemplos de algoritmos recursivos.

Imagine que temos uma longa lista de números armazenada num vetor v. O espaço reservado para o vetor pode ter sido criado pela declaração

```
int v[N];
```

sendo `N` uma constante (possivelmente definida por um `#define`, conforme Seção J.4). Se a lista de números está armazenada nas posições $0, 1, \ldots, n-1$ do vetor, diremos que

$$v[0\,..\,n-1]$$

é um vetor de inteiros. É claro que devemos ter $0 \leq n \leq$ `N`. Se $n = 0$, o vetor $v[0\,..\,n-1]$ está **vazio**. Se $n =$ `N`, o vetor está **cheio**.

3.1 Busca

Dado um inteiro x e um vetor de inteiros $v[0\,..\,n-1]$, considere o problema de encontrar um índice k tal que $v[k] = x$. O problema faz sentido com qualquer $n \geq 0$. Se $n = 0$, o vetor é vazio e portanto essa instância do problema não tem solução.

[1] O aspecto mais importante de uma sequência é a ordem de seus elementos: há um *primeiro* elemento, um *segundo* elemento etc., um *último* elemento (todas as sequências neste livro são finitas). Cada elemento da sequência, exceto o último, tem um sucessor.

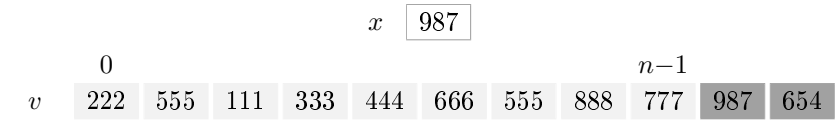


Figura 3.1: Problema da busca: encontrar k tal que $0 \le k < n$ e $v[k] = x$.

É preciso começar com uma decisão de projeto: que fazer se x não estiver no vetor? Adotaremos a convenção de devolver -1 nesse caso. A convenção é razoável pois -1 não pertence ao conjunto $0..n-1$ de índices "válidos". Para implementar esta convenção, convém varrer o vetor do fim para o começo:

```
/* Esta função recebe um número x e um vetor v[0..n-1]
 * com n >= 0 e devolve k no intervalo 0..n-1 tal que
 * v[k] == x. Se tal k não existe, devolve -1. */
int Busca (int x, int v[], int n) {
   int k;
   k = n - 1;
   while (k >= 0 && v[k] != x)
      k -= 1;
   return k;
}
```

Observe como o algoritmo é eficiente e elegante. Ele funciona corretamente mesmo quando o vetor está vazio, ou seja, quando n vale 0.

Maus exemplos. Para fazer contraste com o código acima, seguem algumas amostras de código deselegante, ineficiente e incorreto. A primeira, muito popular, usa uma variável booleana[3] sem necessidade:

```
int k = n - 1, achou = 0;
while (k >= 0 && achou == 0) {    /* deselegante */
   if (v[k] == x) achou = 1;      /* deselegante */
   else k -= 1;
}
return k;
```

[2] Em C, a expressão `k -= 1` é uma abreviatura de `k = k-1`.

[3] Uma variável é **booleana** se admite apenas dois valores: 0 e 1.

A segunda, popular mas deselegante, trata do vetor vazio em separado:

```
int k;
if (n == 0) return -1;    /* deselegante */
k = n - 1;
while (k >= 0 && v[k] != x) k -= 1;
return k;
```

A terceira é ineficiente, pois continua calculando depois de ter encontrado uma solução, e deselegante, pois inicializa uma variável desnecessariamente:

```
int k = 0;                    /* deselegante */
int sol = -1;
for (k = n-1; k >= 0; k--)    /* ineficiente */
   if (v[k] == x) sol = k;
return sol;
```

Na amostra seguinte, a última iteração comete o erro de examinar `v[-1]` porque a ordem dos termos na expressão que controla o `while` está errada (veja o Seção J.2):

```
int k = n - 1;
while (v[k] != x && k >= 0)    /* errado! */
   k -= 1;
return k;
```

Exercícios

3.1.1 Qual o invariante (veja Seção 1.2) do processo iterativo na função `Busca`?

3.1.2 Analise a seguinte variante[4] do código da função `Busca`.

```
int k;
for (k = n-1; k >= 0; k--)
   if (v[k] == x) return k;
return -1;
```

3.1.3 Critique a seguinte versão da função `Busca`:

```
int k = 0;
while (k < n && v[k] != x) k += 1;[5]
if (v[k] == x) return k;
else return -1;
```

3.1.4 Critique a seguinte versão da função `Busca`:

[4] Não confunda *variante* com *invariante*...

[5] A expressão `k += 1` é uma abreviatura de `k = k+1`.

```
   int sol;
   for (k = 0; k < n; k++) {
      if (v[k] == x) sol = k;
      else sol = -1; }
   return sol;
```

3.1.5 Tome uma decisão de projeto diferente da adotada no texto: se x não estiver em $v[0\,..\,n{-}1]$, a função deve devolver n. Escreva a versão correspondente da função `Busca`. Para evitar o grande número de comparações de k com n, coloque uma "sentinela" em $v[n]$.

3.1.6 Considere o problema de determinar o valor de um elemento máximo de um vetor `v[0..n-1]`. A seguinte função resolve o problema?

```
int máximo (int n, int v[]) {
   int j, x;
   x = v[0];
   for (j = 1; j < n; j += 1)
      if (x < v[j]) x = v[j];
   return x; }
```

Faz sentido trocar "`x = v[0]`" por "`x = 0`"? Faz sentido trocar "`x = v[0]`" por "`x = INT_MIN`"? [6] Faz sentido trocar "`x < v[j]`" por "`x <= v[j]`"?

3.2 Busca recursiva

A função `Busca` da seção anterior pode ser reescrita em estilo recursivo.[7] A ideia do código é simples: se $n = 0$ então o vetor é vazio e portanto x não está em $v[0\,..\,n{-}1]$; se $n > 0$ então x está em $v[0\,..\,n{-}1]$ se e somente se $x = v[n{-}1]$ ou está x no vetor $v[0\,..\,n{-}2]$.

```
/* Recebe x, v e n >= 0 e devolve k tal que 0 <= k < n
 * e v[k] == x. Se tal k não existe, devolve -1. */
int BuscaR (int x, int v[], int n) {
   if (n == 0) return -1;
   if (x == v[n-1]) return n - 1;
   return BuscaR (x, v, n - 1);
}
```

Mau exemplo. A seguinte versão da função `BuscaR` é muito deselegante. Ela coloca o caso $n = 1$ na base da recursão e portanto só funciona se $n \geq 1$.

[6] `INT_MIN` é o valor do menor número do tipo `int`. Veja Apêndice C e Seção K.5.

[7] A versão recursiva desta função pode não ser uma alternativa muito prática para a versão iterativa pois a pilha de recursão (veja Seção 6.5) consome memória.

```
int feio (int x, int v[], int n) {
   if (n == 1) {                    /* deselegante */
      if (x == v[0]) return 0;      /* deselegante */
      else return -1;
   }
   if (x == v[n-1]) return n - 1;
   return feio (x, v, n - 1);
}
```

Exercícios

3.2.1 Critique a seguinte função. Ela promete decidir se `x` está em `v[0..n-1]`, devolvendo `1` em caso afirmativo e `0` em caso negativo.

```
int muitofeio (int x, int v[], int n) {
   if (n == 0) return 0;
   else {
      int achei;
      achei = muitofeio (x, v, n - 1);  /* ineficiente */
      if (achei || x == v[n-1]) return 1;
      else return 0; } }
```

3.2.2 O autor da função abaixo afirma que ela decide se `x` está no vetor `v[0..n-1]`. Critique o código.

```
int busc (int x, int v[], int n) {
   if (v[n-1] == x) return 1;
   else return busc (x, v, n - 1); }
```

3.2.3 Escreva um programa para testar o funcionamento da função `BuscaR`. O seu programa deve gerar um vetor aleatório (veja Apêndice I) para fazer o teste. Acrescente ao seu programa uma função que *confira* a resposta dada por `BuscaR`.

3.3 Remoção

A operação de remoção consiste em retirar do vetor $v[0\,..\,n{-}1]$ o elemento que tem índice k e fazer com que o vetor resultante tenha índices $0, 1, \ldots, n{-}2$. (Por exemplo, o resultado da remoção do elemento de índice 3 no vetor $000, 111, 222, 333, 444, 555$ é o vetor $000, 111, 222, 444, 555$.) É claro que a operação só faz sentido se $0 \leq k < n$. A seguinte função faz a remoção e devolve o número de elementos do vetor resultante:

```
/* Remove o elemento de índice k do vetor v[0..n-1] e
 * devolve o novo valor de n. A função supõe 0 <= k < n. */
```

```
int Remove (int k, int v[], int n) {
   int j;
   for (j = k; j < n-1; j++)
      v[j] = v[j+1];
   return n - 1;
}
```

Note como tudo funciona bem mesmo quando $k = n - 1$ ou $k = 0$. Para remover o elemento de índice `51` (supondo que `51` $< n$), basta dizer

```
n = Remove (51, v, n);
```

Versão recursiva. É um bom exercício escrever uma versão recursiva da função `Remove`. O tamanho de uma instância do problema é medido pela diferença $n - k$ e o problema é considerado pequeno se $n - k = 1$:

```
int RemoveR (int k, int v[], int n) {
   if (k == n-1) return n - 1;
   else {
      v[k] = v[k+1];
      return RemoveR (k + 1, v, n);
   }
}
```

Exercícios

3.3.1 Critique a seguinte variante da parte central do código da função `Remove`:

```
for (j = n-2; j >= k; j--) v[j] = v[j+1];
```

3.3.2 Critique a seguinte variante da parte central do código da função `Remove`:

```
for (j = k; j < n-1; j++) v[j] = v[j+1];
v[n-1] = 0;
```

3.3.3 Critique a seguinte variante da parte central do código da função `Remove`:

```
if (k < n-1)
   for (j = k; j < n-1; j++) v[j] = v[j+1];
```

3.3.4 Discuta a seguinte versão da função `RemoveR`:

```
int removeR2 (int k, int v[], int n) {
   if (k < n - 1) {
      removeR2 (k, v, n - 1);
      v[n-2] = v[n-1]; }
   return n - 1; }
```

3.4 Inserção

A operação de inserção consiste em introduzir um novo elemento y entre a posição de índice $k-1$ e a posição de índice k no vetor $v[0\,..\,n-1]$. Isto faz sentido não só quando $1 \le k \le n-1$, mas também quando k é igual a 0 (insere no início) e quando k é igual a n (insere no fim). Em suma, faz sentido quando $0 \le k \le n$.

```
/* Insere y entre as posições k-1 e k do vetor v[0..n-1]
 * e devolve o novo valor de n. Supõe que 0 <= k <= n.
int Insere (int k, int y, int v[], int n) {
   int j;
   for (j = n; j > k; j--)
      v[j] = v[j-1];
   v[k] = y;
   return n + 1;
}
```

A função só deve ser usada se o vetor não estiver cheio, ou seja, se $n <$ `N` (veja página 11). Para inserir um novo elemento com valor `999` entre as posições `50` e `51` (supondo que `51` $\le n$), basta dizer

```
n = Insere(51, 999, v, n);
```

Versão recursiva. É um bom exercício escrever uma versão recursiva da função `Insere`:

```
int InsereR (int k, int y, int v[], int n) {
   if (k == n) v[n] = y;
   else {
      v[n] = v[n-1];
      InsereR (k, y, v, n - 1);
   }
   return n + 1;
}
```

Exercícios

3.4.1 Critique a seguinte versão da função `InsereR`:

```
int insereR2 (int k, int y, int v[], int n) {
   if (k == n) {
```

```
      v[n] = y;
      return n + 1;
   } else {
      v[n] = v[n-1];
      return insereR2 (k, y, v, n - 1); } }
```

3.4.2 Discuta a seguinte variante da função `InsereR`:

```
int insereR3 (int k, int y, int v[], int n) {
   if (k == n) {
      v[n] = y;
      return n + 1;
   } else {
      int z = v[k];
      v[k] = y;
      return insereR3 (k + 1, z, v, n); } }
```

3.5 Busca seguida de remoção

Considere agora uma combinação das operações de busca e remoção. Suponha que queremos remover todos os elementos nulos do vetor $v[0\,..\,n{-}1]$. Aplicada ao vetor

$$111, 222, 0, 0, 333, 0, 444, 0\,,$$

por exemplo, a função deve produzir $111, 222, 333, 444$. É claro que a função deve devolver o novo valor de n.

```
/* Esta função remove todos os elementos nulos de v[0..n-1],
 * deixa o resultado em v[0..i-1], e devolve i. */
int RemoveZeros (int v[], int n) {
   int i = 0, j;
   for (j = 0; j < n; j++)
      if (v[j] != 0) {
         v[i] = v[j];
         i += 1;
      }
   return i;
}
```

Observe como o código funciona bem nos casos extremos: quando n vale 0, quando $v[0\,..\,n{-}1]$ não tem zeros, e quando $v[0\,..\,n{-}1]$ só tem zeros. No início de cada iteração, temos os seguintes invariantes: $i \leq j$ e $v[0\,..\,i{-}1]$ é o resultado da remoção dos zeros do vetor $v[0\,..\,j{-}1]$ original.

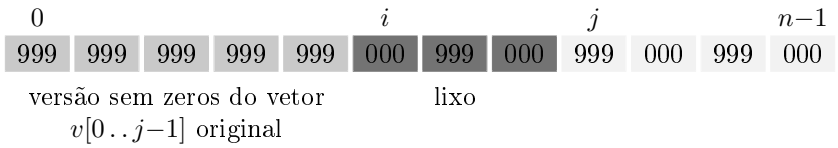


Figura 3.2: Estado do vetor $v[0..n-1]$ no início de uma iteração da função `RemoveZeros`.

Maus exemplos. É fácil escrever uma versão ineficiente de `RemoveZeros`. O código abaixo é simples mas desperdiça tempo e espaço (além de depender da restrição artificial "$n \leq 1000$"):

```
int w[1000], i = 0, j;
for (j = 0; j < n; j++) w[j] = v[j];  /* ineficiente */
for (j = 0; j < n; j++)
   if (w[j] != 0)
      v[i] = w[j], i += 1;
return i;
```

Eis outra solução deselegante e ineficiente. O comando `v[j] = v[j+1]` não copia `v[j+1]` para o seu lugar definitivo e por isso precisa ser repetido muitas vezes:

```
int i = 0, j = 0;     /* "j = 0" é supérfluo */
while (i < n) {
   if (v[i] != 0) i += 1;
   else {
      for (j = i; j+1 < n; j++)   /* ineficiente */
         v[j] = v[j+1];           /* ineficiente */
      --n;
   }
}
return n;
```

Versão recursiva. Eis uma solução recursiva do problema:

```
int RemoveZerosR (int v[], int n) {
   int m;
   if (n == 0) return 0;
   m = RemoveZerosR (v, n - 1);
   if (v[n-1] == 0) return m;
   v[m] = v[n-1];
   return m + 1;
}
```

Exercícios

3.5.1 Critique a seguinte variante da função `RemoveZeros`:

```
int i, j;
for (i = n-1; i >= 0; i--)
   if (v[i] == 0) {
      for (j = i; j < n-1; j++) v[j] = v[j+1];
      --n; }
return n;
```

3.5.2 Critique a seguinte versão da função `RemoveZeros`:

```
int i, j;
for (i = 0; i < n; i++)
   if (v[i] == 0) {
      for (j = i; j+1 < n; j++)  v[j] = v[j+1];
      n -= 1; }
return n;
```

3.5.3 Critique a seguinte versão da função `RemoveZeros`. Há alguma maneira simples de corrigir o código?

```
int i, z = 0;
for (i = 0; i < n; i++) {
   if (v[i] == 0) z += 1;
   v[i-z] = v[i]; }
return n - z;
```

Capítulo 4

Listas encadeadas

Uma lista encadeada é uma representação de uma sequência de objetos na memória do computador. Cada elemento da sequência é armazenado em uma "célula" da lista. As células que armazenam elementos consecutivos da sequência não ficam necessariamente em posições consecutivas da memória.

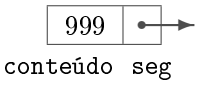
4.1 Definição

Uma **lista encadeada** é uma sequência de registros (veja Apêndice E) que chamaremos **células**. Cada célula contém um objeto de determinado tipo e o endereço (veja Seção D.1) da célula seguinte (no caso da última célula, esse endereço é `NULL`).

Suporemos neste capítulo que os objetos armazenados nas células são do tipo `int`. A estrutura das células pode, então, ser definida assim:

```
struct cel {
   int         conteúdo;
   struct cel *seg;
};
```

[1]



É conveniente tratar as células como um novo tipo de dados (veja Seção J.3), que chamaremos `célula`:

```
typedef struct cel célula;
```

Uma célula `c` e um ponteiro `p` para uma célula podem agora ser declarados

[1] Veja Seção A.4.

assim:

```
célula  c;
célula *p;
```

Se `c` é uma célula então `c.conteúdo` é o conteúdo da célula e `c.seg` é o endereço da célula seguinte. Se `p` é o endereço de uma célula, então `p->conteúdo` é o conteúdo da célula e `p->seg` é o endereço da célula seguinte (veja Seção E.2). Se `p` é o endereço da última célula da lista então `p->seg` vale `NULL`.

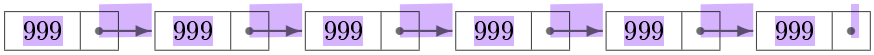


Figura 4.1: Uma lista encadeada (veja Figura D.2 no Apêndice D).

O **endereço** de uma lista encadeada é o endereço de sua primeira célula. Se `p` é o endereço de uma lista, podemos dizer, simplesmente, "`p` é uma lista" e "considere a lista `p`". Reciprocamente, a expressão "`p` é uma lista" deve ser interpretada como "`p` é o endereço da primeira célula de uma lista".

A seguinte observação coloca em evidência a natureza recursiva das listas encadeadas: para toda lista encadeada `p`,

1. `p` é `NULL` ou
2. `p->seg` é uma lista encadeada.

Muitos algoritmos que manipulam listas ficam mais simples quando escritos em estilo recursivo.

Exercício

4.1.1 Dizemos que uma `célula D` é *sucessora* de uma `célula C` se `C.seg` = `&D`. Nas mesmas condições, dizemos que `C` é *antecessora* de `D`. Um *ciclo* é uma sequência $(C_1, C_2, \ldots, C_k)$ de `células` tal que C_{i+1} é sucessora de C_i para $i = 1, 2, \ldots, k-1$ e C_1 é sucessora de C_k. Mostre que uma coleção $\mathcal{L}$ de `células` é uma lista encadeada se e somente se (1) a sucessora de cada elemento de $\mathcal{L}$ está em $\mathcal{L}$, (2) cada elemento de $\mathcal{L}$ tem no máximo uma antecessora, (3) exatamente um elemento de $\mathcal{L}$ não tem antecessora em $\mathcal{L}$ e (4) não há ciclos em $\mathcal{L}$.

4.2 Listas com cabeça e sem cabeça

Uma lista encadeada pode ser vista de duas maneiras diferentes, dependendo do papel que sua primeira célula desempenha. Na lista **com cabeça**, a primeira célula serve apenas para marcar o início da lista e portanto o seu conteúdo é irrelevante. A primeira célula é a **cabeça** da lista. Se `lst` é o endereço da cabeça então `lst->seg` vale `NULL` se e somente se a lista está vazia. Para criar uma lista vazia deste tipo, basta dizer

```
célula c, *lst;              célula *lst;
c.seg = NULL;        ou      lst = malloc (sizeof (célula));
lst = &c;                    lst->seg = NULL;
```

[2]

Na lista **sem cabeça**, o conteúdo da primeira célula é tão relevante quanto o das demais. A lista está vazia se não tem célula alguma. Para criar uma lista vazia `lst` basta dizer

```
célula *lst;
lst = NULL;
```

Embora as listas sem cabeça sejam mais "puras", trataremos preferencialmente de listas *com* cabeça, pois elas são mais fáceis de manipular. O caso das listas sem cabeça será relegado aos exercícios.

```
void Imprima (célula *lst) {
   célula *p;
   for (p = lst; p != NULL; p = p->seg)
      printf ("%d\n", p->conteúdo);
}
```

Figura 4.2: A função imprime o conteúdo de uma lista encadeada `lst` sem cabeça. Para aplicar a função a uma lista *com* cabeça, diga `Imprima (lst->seg)`. Outra possibilidade é trocar o fragmento "`for (p = lst`" por "`for (p = lst->seg`", obtendo assim uma versão da função que só se aplica a listas com cabeça.

4.3 Busca em lista encadeada

É fácil verificar se um objeto x pertence a uma lista encadeada, ou seja, se x é igual ao conteúdo de alguma célula da lista:

[2] Veja Seção F.2.

```
/* Esta função recebe um inteiro x e uma lista encadeada lst
 * com cabeça. Devolve o endereço de uma célula que contém x
 * ou devolve NULL se tal célula não existe. */
célula *Busca (int x, célula *lst) {
   célula *p;
   p = lst->seg;
   while (p != NULL && p->conteúdo != x)
      p = p->seg;
   return p;
}
```

(Observe como o código é simples. Observe como produz o resultado correto mesmo quando a lista está vazia.) Eis uma versão recursiva da função:

```
célula *BuscaR (int x, célula *lst) {
   if (lst->seg == NULL)
      return NULL;
   if (lst->seg->conteúdo == x)
      return lst->seg;
   return BuscaR (x, lst->seg);
}
```

Exercícios

4.3.1 Que acontece se trocarmos "`while (p != NULL && p->conteúdo != x)`" por "`while (p->conteúdo != x && p != NULL)`" na função `Busca`?

4.3.2 Critique a seguinte variante da função `Busca`:

```
int achou = 0;
célula *p;
p = lst->seg;
while (p != NULL && achou != 0) {
   if (p->conteúdo == x) achou = 1;
   p = p->seg; }
if (achou) return p;
else return NULL;
```

4.3.3 Lista sem cabeça. Escreva uma versão da função `Busca` para listas sem cabeça.

4.3.4 Mínimo. Escreva uma função que encontre uma célula de conteúdo mínimo. Faça duas versões: uma iterativa e uma recursiva.

4.3.5 Lista crescente. Uma lista é *crescente* se o conteúdo de cada célula não é maior que o conteúdo da célula seguinte. Escreva uma função que faça uma busca

em uma lista crescente. Faça versões para listas com e sem cabeça. Faça uma versão recursiva e outra iterativa.

4.4 Remoção de uma célula

Suponha que queremos remover uma célula de uma lista. Como devemos especificar a célula a ser removida? Parece natural apontar para a célula em questão, mas é fácil perceber o defeito da ideia. É melhor apontar para a célula *anterior* à que queremos remover. (É bem verdade que esta convenção não permite remover a primeira célula da lista, mas esta operação não é necessária no caso de listas com cabeça.) A função abaixo implementa a ideia:

```
/* Esta função recebe o endereço p de uma célula em uma
 * lista encadeada e remove da lista a célula p->seg.
 * A função supõe que p != NULL e p->seg != NULL. */
void Remove (célula *p) {
   célula *lixo;
   lixo = p->seg;
   p->seg = lixo->seg;
   free (lixo);[3]
}
```

A função não faz mais que alterar o valor de um ponteiro. Não é preciso copiar coisas de um lugar para outro, como fizemos na Seção 3.3 ao remover um elemento de um vetor. A função consome sempre o mesmo tempo, quer a célula a ser removida esteja perto do início da lista, quer esteja perto do fim.

Exercícios

4.4.1 Critique a seguinte variante da função `Remove`:

```
void Remove (célula *p, célula *lst) {
   célula *lixo;
   lixo = p->seg;
   if (lixo->seg == NULL) p->seg = NULL;
   else p->seg = lixo->seg;
   free (lixo); }
```

4.4.2 Lista sem cabeça. Escreva uma função que remova uma determinada célula de uma lista encadeada sem cabeça. (Será preciso tomar algumas decisões de projeto antes de começar a programar.)

[3] Veja Seção F.3.

4.5 Inserção de nova célula

Suponha que queremos inserir uma nova célula com conteúdo y entre a célula apontada por `p` e a seguinte. É claro que isso só faz sentido se `p` for diferente de `NULL`.

```
/* A função insere uma nova célula em uma lista encadeada
 * entre a célula p e a seguinte (supõe-se que p != NULL).
 * A nova célula terá conteúdo y. */
void Insere (int y, célula *p) {
   célula *nova;
   nova = malloc (sizeof (célula));
   nova->conteúdo = y;
   nova->seg = p->seg;
   p->seg = nova;
}
```

A função não faz mais que alterar os valores de alguns ponteiros. Não há movimentação de células para "abrir espaço" para um nova célula, como fizemos na Seção 3.4 ao inserir um novo elemento em um vetor. Assim, o tempo que a função consome não depende do ponto de inserção: tanto faz inserir uma nova célula na parte inicial da lista quanto na parte final.

A função se comporta corretamente mesmo quando a inserção se dá no fim da lista, isto é, quando `p->seg` vale `NULL`. Se a lista tem cabeça, a função pode ser usada para inserir no início da lista: basta que `p` aponte para a célula-cabeça. Mas a função não é capaz de inserir antes da primeira célula de uma lista sem cabeça.

Exercícios

4.5.1 Por que a seguinte versão de `Insere` não funciona?

```
célula nova;
nova.conteúdo = y;
nova.seg = p->seg;
p->seg = &nova;
```

4.5.2 Escreva uma função que insira uma nova célula entre a célula cujo endereço é `p` e a *anterior*.

4.5.3 Lista sem cabeça. Escreva uma função que insira uma nova célula numa dada posição de uma lista encadeada sem cabeça. (Será preciso tomar algumas decisões de projeto antes de começar a programar.)

4.6 Busca seguida de remoção ou inserção

Considere uma lista encadeada com cabeça. Dado um inteiro x, queremos remover da lista a primeira célula que contiver x; se tal célula não existe, não é preciso fazer nada.

```
/* Esta função recebe uma lista encadeada lst com cabeça
 * e remove da lista a primeira célula que contiver x,
 * se tal célula existir. */
void BuscaERemove (int x, célula *lst) {
   célula *p, *q;
   p = lst;
   q = lst->seg;
   while (q != NULL && q->conteúdo != x) {
      p = q;
      q = q->seg;
   }
   if (q != NULL) {
      p->seg = q->seg;
      free (q);
   }
}
```

No início de cada iteração, imediatamente antes da comparação de `q` com `NULL`, vale a relação `q` = `p->seg` (ou seja, `q` está sempre um passo à frente de `p`).

Suponha agora que queremos inserir na lista uma nova célula com conteúdo y imediatamente antes da primeira célula que tiver conteúdo x; se tal célula não existe, devemos inserir y no fim da lista.

```
/* Recebe uma lista encadeada lst com cabeça e insere uma
 * nova célula na lista imediatamente antes da primeira que
 * contiver x. Se nenhuma célula contiver x, a nova célula
 * será inserida no fim da lista. A nova célula terá
 * conteúdo y. */
void BuscaEInsere (int y, int x, célula *lst) {
   célula *p, *q, *nova;
   nova = malloc (sizeof (célula));
   nova->conteúdo = y;
   p = lst;
   q = lst->seg;
```

```
    while (q != NULL && q->conteúdo != x) {
       p = q;
       q = q->seg;
    }
    nova->seg = q;
    p->seg = nova;
}
```

Exercícios

4.6.1 Escreva uma versão da função `BuscaERemove` para listas encadeadas sem cabeça. (Veja Exercício 4.4.2.)

4.6.2 Escreva uma versão da função `BuscaEInsere` para listas encadeadas sem cabeça. (Veja Exercício 4.5.3.)

4.6.3 Escreva uma função para remover de uma lista encadeada todos os elementos que contêm x. Faça uma versão iterativa e uma recursiva.

4.6.4 Escreva uma função que remova de uma lista encadeada uma célula cujo conteúdo tem valor mínimo. Faça uma versão iterativa e uma recursiva.

4.7 Exercícios: manipulação de listas

A maioria dos exercícios desta seção tem duas versões: uma para lista com cabeça e outra para lista sem cabeça. Além disso, é interessante resolver cada exercício de duas maneiras: uma iterativa e uma recursiva.

4.7.1 Vetor para lista. Escreva uma função que copie um vetor para uma lista encadeada.

4.7.2 Lista para vetor. Escreva uma função que copie uma lista encadeada para um vetor.

4.7.3 Cópia. Escreva uma função que faça uma cópia de uma lista dada.

4.7.4 Comparação. Escreva uma função que decida se duas listas dadas têm o mesmo conteúdo.

4.7.5 Concatenação. Escreva uma função que concatene duas listas encadeadas (isto é, "amarre" a segunda no fim da primeira).

4.7.6 Contagem. Escreva uma função que conte o número de células de uma lista encadeada.

4.7.7 Ponto médio. Escreva uma função que receba uma lista encadeada e devolva o endereço de uma célula que esteja o mais próximo possível do ponto médio da lista. Faça isso sem calcular explicitamente o número n de células da lista e o quociente $n/2$.

4.7.8 Contagem e remoção. Escreva uma função que remova a k-ésima célula de uma lista encadeada.

4.7.9 Contagem e inserção. Escreva uma função que insira uma nova célula com conteúdo x entre a k-ésima e a $(k+1)$-ésima células de uma lista encadeada.

4.7.10 Liberação. Escreva uma função que aplique a função `free` a todas as células de uma lista encadeada. Estamos supondo, é claro, que cada célula da lista foi originalmente alocado por `malloc`.

4.7.11 Inversão. Escreva uma função que inverta a ordem das células de uma lista encadeada (a primeira passa a ser a última, a segunda passa a ser a penúltima etc.). Faça isso sem criar novas células; apenas altere os ponteiros.

4.7.12 Projeto de programação. Digamos que um *documento* é um vetor de caracteres contendo apenas letras, espaços e sinais de pontuação. Digamos que uma *palavra* é um segmento maximal que consiste apenas de letras. Escreva uma função que imprima uma relação de todas as palavras de um documento dado juntamente com o número de ocorrências de cada palavra.

4.8 Outros tipos de listas encadeadas

Poderíamos definir vários outros tipos de listas encadeadas além do tipo básico discutido acima. Seguem dois exemplos importantes.

a. Numa lista encadeada **circular**, a última célula aponta para a primeira. A lista pode ou não ter uma célula-cabeça. (Se não tiver cabeça, as expressões "primeira célula" e "última célula" não fazem muito sentido.)

b. Numa lista **duplamente encadeada**, cada célula contém o endereço da célula anterior e o da célula seguinte. A lista pode ou não ter uma célula-cabeça, conforme as conveniências do programador.

As seguintes questões são apropriadas para qualquer tipo de lista encadeada: Em que condições a lista está vazia? Como remover a célula apontada por `p`? Como remover a célula seguinte à apontada por `p`? Como remover a célula anterior à apontada por `p`? Como inserir uma nova célula entre a apontada por `p` e a anterior? Como inserir uma nova célula entre a apontada por `p` e a seguinte?

Exercícios

4.8.1 Descreva, em C, a estrutura de uma célula de uma lista duplamente encadeada.

4.8.2 Escreva uma função que remova de uma lista duplamente encadeada a célula cujo endereço é `p`. Que dados sua função recebe? Que coisa devolve?

4.8.3 Suponha uma lista duplamente encadeada. Escreva uma função que insira uma nova célula com conteúdo y logo após a célula cujo endereço é `p`. Que dados sua função recebe? Que coisa devolve?

4.8.4 PROBLEMA DE JOSEPHUS. Imagine n pessoas dispostas em círculo. Suponha que as pessoas estão numeradas de 1 a n no sentido horário. Começando com a pessoa de número 1, percorra o círculo no sentido horário e elimine cada m-ésima pessoa enquanto o círculo tiver duas ou mais pessoas. (Veja *Josephus problem* na Wikipedia [21].) Qual o número do sobrevivente? Escreva e teste uma função que resolva o problema.

4.8.5 Leia o verbete *Linked list* na Wikipedia [21].

Capítulo 5

Filas

Fila: Fileira de pessoas que se colocam umas atrás das outras,
pela ordem cronológica de chegada a guichês
ou a quaisquer estabelecimentos onde haja grande afluência de interessados.

— *Novo Dicionário Aurélio*

Uma fila é uma sequência dinâmica, isto é, uma sequência da qual elementos podem ser removidos e na qual novos elementos podem ser inseridos. Mais especificamente, uma **fila** é uma sequência de objetos, todos do mesmo tipo, sujeita às seguintes regras de comportamento: (1) sempre que solicitamos a remoção de um elemento, o elemento removido é o primeiro da sequência e (2) sempre que solicitamos a inserção de um novo objeto, o objeto é inserido no fim da sequência.

Podemos resumir o comportamento de uma fila com a seguinte frase: o elemento removido da fila é sempre o que está lá há mais tempo. Outra maneira de dizer isso: o primeiro objeto inserido na fila é também o primeiro a ser removido. Esta política é conhecida pela abreviatura FIFO da expressão *First-In-First-Out*.

5.1 Implementação em vetor

Uma fila pode ser armazenada em um segmento `f[s..t-1]` de um vetor `f[0..N-1]`. É claro que devemos ter $0 \le s \le t \le N$. O primeiro elemento da fila está na posição `s` e o último na posição `t-1`. A fila está **vazia** se `s` é igual a `t` e **cheia** se `t` é igual a `N`. Para **remover** um elemento da fila basta dizer

```
x = f[s++];
```

o que equivale ao par de comandos "`x = f[s]; s += 1;`" (veja Seção J.1). É

claro que o programador não deve fazer isso se a fila estiver vazia. Para **inserir** um objeto `y` na fila basta dizer

```
f[t++] = y;
```

Se o programador fizer isso quando a fila já está cheia, dizemos que a fila transbordou. Em geral, a tentativa de inserir em uma fila cheia é um evento excepcional, que resulta de um mau planejamento lógico do seu programa.

Figura 5.1: O vetor `f[s..t-1]` armazena uma fila.

Exercício

5.1.1 Suponha que, diferentemente da convenção adotada no texto, a parte do vetor ocupada pela fila é `f[s..t]`. Escreva o comando que remove um elemento da fila. Escreva o comando que insere um objeto `y` na fila.

5.2 Aplicação: distâncias em uma rede

Imagine n cidades numeradas de 0 a $n-1$ e interligadas por estradas de mão única. As ligações entre as cidades são representadas por uma matriz `A` (veja Seção F.5) definida da seguinte maneira: `A[`x`][`y`]` vale `1` se existe estrada da cidade x para a cidade y e vale `0` em caso contrário. (Veja Figura 5.2.)

A **distância**[1] de uma cidade o a uma cidade x é o menor número de estradas que é preciso percorrer para ir de o a x. Nosso problema: *determinar a distância de uma dada cidade o a cada uma das outras cidades.*

As distâncias serão armazenadas em um vetor d de tal modo que d`[`x`]` seja a distância de o a x. Se for impossível chegar de o a x, podemos dizer que d`[`x`]` vale ∞. Usaremos -1 para representar ∞ (uma vez que nenhuma distância "real" pode ter valor -1).

O seguinte algoritmo usa o conceito de fila para resolver nosso problema das distâncias. Uma cidade é considerada *ativa* se já foi visitada mas as estradas

[1] A palavra *distância* já traz embutida a ideia de minimalidade. As expressões "distância mínima" e "menor distância" são redundantes.

que nela começam ainda não foram exploradas. O algoritmo mantém as cidades ativas numa fila. Em cada iteração, o algoritmo remove da fila uma cidade x e insere na fila todas as vizinhas a x que ainda não foram visitadas. Eis uma implementação do algoritmo:

```
/* A matriz A representa as interligações entre cidades
 * 0,1,..,n-1: há uma estrada (de mão única) de x a y se
 * e somente se A[x][y] == 1. A função devolve um vetor d
 * tal que d[x] é a distância da cidade o à cidade x. */
int *Distâncias (int **A, int n, int o) {
   int *d, x, y;
   int *f, s, t;
   d = malloc (n * sizeof (int));[2]
   for (x = 0; x < n; x++) d[x] = -1;
   d[o] = 0;
   f = malloc (n * sizeof (int));
   s = 0; t = 1; f[s] = o;
   while (s < t) {
      /* f[s..t-1] é uma fila de cidades */
      x = f[s++];
      for (y = 0; y < n; y++)
         if (A[x][y] == 1 && d[y] == -1) {
            d[y] = d[x] + 1;
            f[t++] = y;
         }
   }
   free (f);
   return d;
}
```

Ao longo da execução do algoritmo, o vetor `f[s..t-1]` armazena a fila de cidades, enquanto `f[0..s-1]` armazena as cidades que já saíram da fila. Para compreender o algoritmo (e provar que ele está correto), basta observar que as seguintes propriedades valem no início de cada iteração, imediatamente antes da comparação "`s < t`":

1. para cada v no vetor `f[0..t-1]`, existe um caminho de o a v, de comprimento `d[`v`]`, cujas cidades estão todas no vetor `f[0..t-1]`;

[2] Veja Seção F.2.

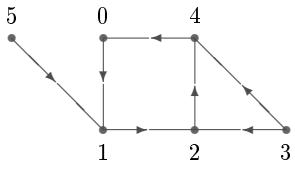
2. para cada v no vetor `f[0..t-1]`, todo caminho de o a v tem comprimento pelo menos d`[v]`;
3. toda estrada que começa em `f[0..s-1]` termina em `f[0..t-1]`.

Deduz-se imediatamente de 1 e 2 que, para cada v no vetor `f[0..t-1]`, o número d`[v]` é a distância de o a v. Para provar que as três propriedades são invariantes, é preciso observar que duas outras propriedades valem no início de cada iteração:

4. d`[f[s]]` $\le$ d`[f[s+1]]` $\le \cdots \le$ d`[f[t-1]]` e
5. d`[f[t-1]]` $\le$ d`[f[s]]` $+ 1$.

Em outras palavras, a sequência de números d`[f[s]]`, ..., d`[f[t-1]]` tem a forma $k, \ldots, k$ ou a forma $k, .., k, k{+}1, .., k{+}1$.

	0	1	2	3	4	5
0	0	1	0	0	0	0
1	0	0	1	0	0	0
2	0	0	0	0	1	0
3	0	0	1	0	1	0
4	1	0	0	0	0	0
5	0	1	0	0	0	0



	0	1	2	3	4	5
d	2	3	1	0	1	6

Figura 5.2: A matriz representa cidades $0, \ldots, 5$ interligadas por estradas de mão única. O vetor d dá as distâncias da cidade 3 a cada uma das demais.

Exercícios

5.2.1 Transbordamento. Na função `Distâncias`, o espaço alocado para o vetor `f` é suficiente? O comando "`f[t++]` = y" pode provocar o transbordamento da fila?

5.2.2 Última iteração. Suponha que os invariantes 1 a 3 valem no início da última iteração da função `Distâncias` (quando `s` é igual a `t`). Mostre que, para cada v no vetor `f[0..t-1]`, o número d`[v]` é a distância de o a v. Mostre também que é impossível ir da cidade o a uma cidade que esteja fora do vetor `f[0..t-1]`.

5.2.3 Primeira iteração. Verifique que os invariantes 1 a 5 valem no início da primeira iteração da função `Distâncias`.

5.2.4 Invariantes. Suponha que os invariantes 1 a 5 da função `Distâncias` valem no início de uma iteração qualquer que não a última. Mostre que elas continuam valendo no início da próxima iteração. (A prova é surpreendentemente longa e delicada.)

5.2.5 Labirinto. Imagine um tabuleiro quadrado 10–por–10. As casas "livres" são

marcadas com 0 e as casas "bloqueadas" com -1. As casas $(1,1)$ e $(10,10)$ estão livres. Ajude uma formiga que está na casa $(1,1)$ a chegar à casa $(10,10)$. Em cada passo, a formiga só pode se deslocar para uma casa livre que esteja à direita, à esquerda, acima ou abaixo da casa em que está.

5.3 Implementação circular

No problema discutido na seção anterior, o vetor que abriga a fila não precisa ter mais componentes que o número total de cidades, pois cada cidade entra na fila no máximo uma vez. Em geral, entretanto, é difícil prever o espaço necessário para abrigar a fila. Nesses casos, é mais seguro implementar a fila de maneira *circular*. Suponha que os elementos da fila estão dispostos no vetor `f[0..N-1]` de uma das seguintes maneiras:

`f[s..t-1]` ou `f[s..N-1] f[0..t-1]`

(veja Figura 5.3). Teremos sempre $0 \le s < N$ e $0 \le t < N$, mas não podemos supor que $s \le t$. A fila está **vazia** se $t = s$ e **cheia** se

$$t+1 = s \quad \text{ou} \quad t+1 = N \ \text{ e } \ s = 0,$$

ou seja, se $(t+1)\,\%\,N = s$.[3] A posição `t` ficará sempre desocupada, para que possamos distinguir uma fila cheia de uma vazia. Para remover um elemento da fila basta fazer

```
x = f[s++];
if (s == N) s = 0;
```

(supondo que a fila não está vazia). Para inserir um objeto `y` na fila (supondo que ela não está cheia), faça

```
f[t++] = y;
if (t == N) t = 0;
```

Exercício

5.3.1 Considere a manipulação de uma fila circular. Escreva uma função que devolva o tamanho da fila. Escreva uma função que remova um elemento da fila e devolva esse elemento; se a fila estiver vazia, não faça nada. Escreva uma função que verifique se a fila está cheia e em caso negativo insira um objeto dado na fila. (Lembre-se de que uma fila é um pacote com três objetos: um vetor e dois índices. Não use variáveis globais.)

[3] O valor da expressão `a%b` é o resto da divisão de `a` por `b`, ou seja, $a - b\lfloor a/b \rfloor$.

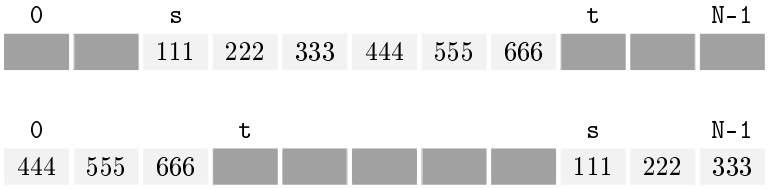


Figura 5.3: Fila circular. Na primeira parte da figura, a fila está armazenada no vetor `f[s..t-1]`. Na segunda parte, a fila está armazenada no vetor `f[s..N-1]` concatenado com `f[0..t-1]`.

5.4 Implementação em lista encadeada

Considere agora a implementação de uma fila em uma lista encadeada. Digamos que as células da lista são do tipo `célula`:

```
typedef struct cel {
   int         valor;
   struct cel *seg;
} célula;
```

É preciso tomar algumas decisões de projeto sobre a maneira de acomodar a fila na lista. Vamos supor que nossa lista encadeada não tem cabeça, que o primeiro elemento da fila ficará na primeira célula e que o último elemento da fila ficará na última célula.

Para manipular a fila, precisamos de dois ponteiros: um ponteiro `s` apontará o primeiro elemento da fila e um ponteiro `t` apontará o último. A fila estará vazia se `s = t = NULL`. Suporemos que `s = NULL` sempre que `t = NULL` e vice-versa. Uma fila vazia pode ser criada assim:

```
célula *s, *t;
s = t = NULL;
```

Para remover um elemento da fila (supondo que ela não está vazia), será preciso passar à função de remoção os *endereços* das variáveis `s` e `t` para que os valores dessas variáveis possam ser alterados:

```
int Remove (célula **es, célula **et) {
   célula *p;
   int x;
   p = *es;
   x = p->valor;
```

```
    *es = p->seg;
    free (p);
    if (*es == NULL) *et = NULL;
    return x;
}
```

A função de inserção precisa levar em conta a possibilidade de inserção em fila vazia:

```
void Insere (int y, célula **es, célula **et) {
    célula *nova;
    nova = malloc (sizeof (célula));
    nova->valor = y;
    nova->seg = NULL;
    if (*et == NULL) *et = *es = nova;
    else {
        (*et)->seg = nova;
        *et = nova;
    }
}
```

Exercícios

5.4.1 Implemente uma fila em uma lista encadeada com célula-cabeça.

5.4.2 Implemente uma fila em uma lista encadeada *circular* com célula-cabeça. O primeiro elemento da fila ficará na segunda célula e o último elemento ficará na célula anterior à cabeça. Para manipular a fila basta conhecer o endereço `ff` da célula-cabeça.

5.4.3 Implemente uma fila em uma lista duplamente encadeada sem célula-cabeça. Mantenha um ponteiro para a primeira célula e um ponteiro para a última.

5.4.4 Leia o verbete *Queue* na Wikipedia [21].

Capítulo 6

Pilhas

Pilha: Porção de objetos dispostos uns sobre os outros.
— *Dicionário Houaiss*

Uma pilha é uma sequência dinâmica, isto é, uma sequência da qual elementos podem ser removidos e na qual novos elementos podem ser inseridos. Mais especificamente, uma **pilha** é uma sequência de objetos, todos do mesmo tipo, sujeita às seguintes regras de comportamento: (1) sempre que solicitamos a remoção de um elemento, o elemento removido é o último da sequência e (2) sempre que solicitamos a inserção de um novo objeto, o objeto é inserido no fim da sequência.

Podemos resumir o comportamento de uma pilha com a seguinte frase: o elemento removido da pilha é sempre o que está lá há menos tempo. Outra maneira de dizer isso: o primeiro objeto inserido na pilha é o último a ser removido. Esta política é conhecida pela abreviatura LIFO da expressão *Last-In-First-Out*.

6.1 Implementação em vetor

Suponha que nossa pilha está armazenada em um vetor `p[0..N-1]`. A parte do vetor efetivamente ocupada pela pilha é `p[0..t-1]`. O índice `t-1` define o **topo** da pilha.

A pilha está **vazia** se `t` vale 0 e **cheia** se `t` vale `N`. Para **remover** um elemento da pilha, ou seja, para **desempilhar** um elemento, faça

```
x = p[--t];
```

o que equivale ao par de comandos "`t -= 1; x = p[t];`" (veja Seção J.1). É claro que o programador não deve fazer isto se a pilha estiver vazia. Para **consultar**

a pilha sem desempilhar basta fazer `x = p[t-1]`. Para **empilhar** um objeto `y`, ou seja, para **inserir** `y` na pilha faça

```
p[t++] = y;
```

o que equivale ao par de comandos "`p[t] = y; t += 1;`". Antes de empilhar, é preciso ter certeza de que a pilha não está cheia. Em geral, a tentativa de inserir em uma pilha cheia é um indício de mau planejamento lógico do seu programa.

Figura 6.1: O vetor `p[0..t-1]` armazena uma pilha.

Exercícios

6.1.1 Suponha que, diferentemente da convenção adotada no texto, a parte do vetor ocupada pela pilha é `p[0..t]`. Escreva o comando que remove um elemento da pilha. Escreva o comando que insere um objeto na pilha.

6.1.2 Inversão de palavras. Escreva uma função que inverta a ordem das letras de cada palavra de uma sentença, preservando a ordem das palavras. Suponha que as palavras da sentença são separadas por espaços. A aplicação da operação à sentença `AMU MEGASNEM ATERCES`, por exemplo, deve produzir `UMA MENSAGEM SECRETA`.

6.1.3 Permutações produzidas pelo desempilhar [10, sec. 2.2.1]. Suponha que os números inteiros $1, 2, 3, 4$ são colocados, nesta ordem, numa pilha inicialmente vazia. Depois de cada operação de empilhar, você pode retirar zero ou mais elementos da pilha. Cada número retirado da pilha é impresso numa folha de papel. Por exemplo, a sequência de operações $\mathsf{E}, \mathsf{E}, \mathsf{D}, \mathsf{E}, \mathsf{D}, \mathsf{D}, \mathsf{E}, \mathsf{D}$, onde E significa "empilhar o próximo número da sequência" e D significa "desempilhar", produz a impressão da sequência $2, 3, 1, 4$. Quais das 24 permutações de $1, 2, 3, 4$ podem ser obtidas desta maneira?

6.2 Aplicação: parênteses e chaves

Considere o problema de decidir se uma dada sequência de parênteses e chaves é bem-formada. Por exemplo, a sequência

```
( ( ) { ( ) } )
```

é bem-formada, enquanto a sequência `( { ) }` é malformada.

Suponha que a sequência de parênteses e chaves está armazenada em uma string `s` (veja Apêndice G). De acordo com as convenções da linguagem C, o último elemento da string é o caractere nulo `'\0'` (veja Apêndice B).

```
/* Esta função devolve 1 se a string s contém uma sequência
 * bem-formada de parênteses e chaves e devolve 0 se
 * a sequência está malformada. */
int BemFormada (char s[]) {
   char *p; int t;
   int n, i;
   n = strlen (s);
   p = malloc (n * sizeof (char));
   t = 0;
   for (i = 0; s[i] != '\0'; i++) {
      /* p[0..t-1] é uma pilha */
      switch (s[i]) {
         case ')': if (t != 0 && p[t-1] == '(') --t;
                   else return 0;
                   break;
         case '}': if (t != 0 && p[t-1] == '{') --t;
                   else return 0;
                   break;
         default: p[t++] = s[i];
      }
   }
   return t == 0;[1]
}
```

(Eu deveria ter invocado `free(p)` antes de cada `return`; só não fiz isso para não obscurecer a lógica da função.) A pilha `p` jamais transborda porque nunca terá mais elementos do que o número, `n`, de caracteres de `s`.

Exercícios

6.2.1 A função `BemFormada` funciona corretamente se `s` tem apenas dois elementos? apenas um? nenhum?

6.2.2 Mostre que o processo iterativo na função `BemFormada` tem o seguinte invariante:

[1] Veja Seção J.2.

no início de cada iteração, a string `s` está bem-formada se e somente se a sequência `p[0..t-1] s[i...]`, formada pela concatenação de `p[0..t-1]` com `s[i...]`, estiver bem-formada.

6.3 Aplicação: notação posfixa

Expressões aritméticas são usualmente escritas em notação *infixa*: os operadores ficam entre os operandos. Na notação *posfixa* (ou *polonesa*) os operadores ficam depois dos operandos. Os exemplos da Figura 6.2 esclarecem o conceito. (A propósito, veja o Exercício 14.2.7.)

notação infixa	notação posfixa
`(A+B*C)`	`ABC*+`
`(A*(B+C)/D-E)`	`ABC+*D/E-`
`(A+B*(C-D*(E-F)-G*H)-I*3)`	`ABCDEF-*-GH*-*+I3*-`
`(A+B*C/D*E-F)`	`ABC*D/E*+F-`
`(A*(B+(C*(D+(E*(F+G))))))`	`ABCDEFG+*+*+*`

Figura 6.2: Expressões aritméticas em notação infixa e notação posfixa. A notação posfixa dispensa parênteses. Os operandos (`A`, `B` etc.) aparecem na mesma ordem nas duas notações.

Nosso problema: traduzir para notação posfixa uma expressão infixa dada. Para simplificar, suporemos que a expressão infixa está correta e contém apenas letras, parênteses e os símbolos `+`, `-`, `*` e `/`. Suporemos também que cada nome de variável tem uma letra apenas. Finalmente, suporemos que a expressão toda está embrulhada em um par de parênteses. Se a expressão está armazenada na string `infix`, o primeiro caractere da string é `'('` e os dois últimos são `')'` e `'\0'`.

Usaremos uma pilha para resolver o problema de tradução. Como a expressão infixa está embrulhada em parênteses, não será preciso preocupar-se com pilha vazia.

```
/* A função abaixo recebe uma expressão infixa infix e
 * devolve a correspondente expressão posfixa. */
char *InfixaParaPosfixa (char infix[]) {
   char *posfix, x;
```

```
   char *p; int t;
   int n, i, j;
   n = strlen (infix);
   posfix = malloc (n * sizeof (char));
   p = malloc (n * sizeof (char));
   t = 0; p[t++] = infix[0];  /* empilha '(' */
   for (j = 0, i = 1; /*X*/ infix[i] != '\0'; i++) {
      /* p[0..t-1] é uma pilha de caracteres */
      switch (infix[i]) {
         case '(': p[t++] = infix[i];  /* empilha */
                   break;
         case ')': while (1) {  /* desempilha */
                      x = p[--t];
                      if (x == '(') break;
                      posfix[j++] = x; }
                   break;
         case '+':
         case '-': while (1) {
                      x = p[t-1];
                      if (x == '(') break;
                      --t;  /* desempilha */
                      posfix[j++] = x; }
                   p[t++] = infix[i];  /* empilha */
                   break;
         case '*':
         case '/': while (1) {
                      x = p[t-1];
                      if (x == '(' || x == '+' || x == '-')
                         break;
                      --t;
                      posfix[j++] = x; }
                   p[t++] = infix[i];
                   break;
         default: posfix[j++] = infix[i]; }
   }
   free (p);
   posfix[j] = '\0';
   return posfix;
}
```

```
infix[0..i-1]   p[0..t-1]  posfix[0..j-1]
(               (
( A             (          A
( A *           ( *        A
( A * (         ( * (      A
( A * (B        ( * (      A B
( A * (B *      ( * ( *    A B
( A * (B * C    ( * ( *    A B C
( A * (B * C +  ( * ( +    A B C *
( A * (B * C + D ( * ( +   A B C * D
( A * (B * C + D) ( *      A B C * D +
( A * (B * C + D))         A B C * D + *
```

Figura 6.3: Resultado da aplicação da função `InfixaParaPosfixa` à expressão infixa `(A*(B*C+D))`. A figura registra os valores das variáveis no início de cada iteração (ou seja, a cada passagem pelo ponto `X` do código). Constantes e variáveis vão diretamente de `infix` para `posfix`. Todo parêntese esquerdo vai para a pilha. Ao encontrar um parêntese direito, a função remove tudo da pilha até o primeiro parêntese esquerdo que encontrar. Ao encontrar `+` ou `-`, a função desempilha tudo até encontrar um parêntese esquerdo. Ao encontrar `*` ou `/`, desempilha tudo até um parêntese esquerdo ou um `+` ou um `-`.

Exercícios

6.3.1 Aplique à expressão infixa `(A+B)*D+E/(F+A*D)+C` o algoritmo de conversão para notação posfixa.

6.3.2 Na função `InfixaParaPosfixa`, suponha que a string `infix` tem n caracteres (sem contar o caractere nulo final). Que altura a pilha pode atingir, no pior caso? Em outras palavras, qual o valor máximo da variável `t`? Que acontece se o número de parênteses for limitado (menor que 10, por exemplo)?

6.3.3 Reescreva o código da função `InfixaParaPosfixa` de maneira um pouco mais compacta, sem os "`while (1)`". Tire proveito dos recursos sintáticos da linguagem C.

6.3.4 Reescreva a função `InfixaParaPosfixa` sem supor que a expressão infixa está embrulhada em um par de parênteses.

6.3.5 Reescreva a função `InfixaParaPosfixa` supondo que a expressão infixa pode estar incorreta.

6.3.6 Reescreva a função `InfixaParaPosfixa` supondo que a expressão pode ter parênteses e chaves.

6.3.7 Valor de expressão posfixa. Suponha dada uma expressão aritmética em notação posfixa sujeita às seguintes restrições: cada variável consiste em uma única letra do conjunto `A..Z`; não há constantes; os únicos operadores são `+`, `-`, `*`, `/` (todos exigem *dois* operandos). Suponha dado também um vetor inteiro `val`, indexado por

`A..Z`, que dá os valores das variáveis. Escreva uma função que calcule o valor da expressão. Cuidado com divisões por zero.

6.4 Implementação em lista encadeada

Uma pilha pode ser implementada em uma lista encadeada. Digamos que as células da lista são do tipo `célula`:

```
typedef struct cel {
   int         valor;
   struct cel *seg;
} célula;
```

Suporemos que nossa lista tem uma célula-cabeça e que o topo da pilha está na segunda célula (e não na última). Uma pilha (vazia) pode ser criada assim:

```
célula cabeça;
célula *p;
p = &cabeça;
p->seg = NULL;
```

Para manipular a pilha, basta dispor do ponteiro `p`, cujo valor será sempre `&cabeça`. A pilha estará **vazia** se `p->seg` for `NULL`. Eis a função que insere um número `y` na pilha:

```
void Empilha (int y, célula *p) {
   célula *nova;
   nova = malloc (sizeof (célula));
   nova->valor = y;
   nova->seg = p->seg;
   p->seg = nova;
}
```

Eis uma função que remove um elemento de uma pilha não vazia:

```
int Desempilha (célula *p) {
   int x; célula *q;
   q = p->seg;
   x = q->valor;
   p->seg = q->seg;
   free (q);
   return x;
}
```

Exercícios

6.4.1 Implemente uma pilha em uma lista encadeada *sem* célula-cabeça. A pilha será especificada pelo endereço da primeira célula da lista.

6.4.2 Reescreva as funções `BemFormada` e `InfixaParaPosfixa` (Seções 6.2 e 6.3 respectivamente) armazenando a pilha em uma lista encadeada.

6.4.3 Leia o verbete *Stack (data structure)* na Wikipedia [21].

6.5 A pilha de execução de um programa

Todo programa C é composto por uma ou mais funções, sendo `main` a primeira função a ser executada. Para executar um programa, o computador usa uma "pilha de execução". A operação pode ser descrita conceitualmente da seguinte maneira. Ao encontrar a invocação de uma função, o computador cria um novo "espaço de trabalho", que contém todos os parâmetros e todas as variáveis locais da função. Esse espaço de trabalho é colocado na pilha de execução (sobre o espaço de trabalho que invocou a função) e a execução da função começa (confinada ao seu espaço de trabalho). Quando a execução da função termina, o seu espaço de trabalho é retirado da pilha e descartado. O espaço de trabalho que estiver agora no topo da pilha é reativado e a execução é retomada do ponto em que havia sido interrompida.

Considere o seguinte exemplo:

```
int G (int a, int b) {
   return a + b;
}

int F (int i, int j, int k) {
   int x;
   x = /*2*/ G (i, j) /*3*/;
   return x + k;
}

int main (void) {
   int i, j, k, y;
   i = 111; j = 222; k = 444;
   y = /*1*/ F (i, j, k) /*4*/;
   printf ("%d\n", y);
   return EXIT_SUCCESS;
}
```

[2]

[2] Veja Seção K.1.

O programa é executado da seguinte maneira:

1. Um espaço de trabalho é criado para a função `main` e colocado na pilha de execução. O espaço contém as variáveis locais `i`, `j`, `k` e `y`. A execução de `main` começa.
2. No ponto `1`, a execução de `main` é suspensa e um espaço de trabalho para a função `F` é colocado na pilha. Esse espaço contém os parâmetros `i`, `j`, `k` da função (com valores `111`, `222` e `444` respectivamente) e a variável local `x`. Começa então a execução de `F`.
3. No ponto `2`, a execução de `F` é suspensa e um espaço de trabalho para a função `G` é colocado na pilha. Esse espaço contém os parâmetros `a` e `b` da função (com valores `111` e `222` respectivamente). Em seguida, começa a execução de `G`.
4. Quando a execução de `G` termina, a função devolve `333`. O espaço de trabalho de `G` é removido da pilha e descartado. O espaço de trabalho de `F` (que agora está no topo da pilha de execução) é reativado e a execução é retomada no ponto `3`. A primeira instrução executada é "`x = 333;`".
5. Quando a execução de `F` termina, a função devolve `777`. O espaço de trabalho de `F` é removido da pilha e descartado. O espaço de trabalho de `main` é reativado e a execução é retomada no ponto `4`. A primeira instrução executada é "`y = 777;`".

No nosso exemplo, `F` e `G` são funções distintas. Mas tudo funcionaria da mesma maneira se `F` e `G` fossem idênticas, ou seja, se `F` fosse uma função recursiva.

Exercício

6.5.1 Escreva uma função iterativa que simule o comportamento da função recursiva abaixo. Use uma pilha.

```
int TTT (int x[], int n) {
   if (n == 0) return 0;
   if (x[n] > 0) return x[n] + TTT (x, n - 1);
   else return TTT (x, n - 1); }
```

Capítulo 7

Busca em vetor ordenado

"Binary search is to algorithms what a wheel is to mechanics:
It is simple, elegant, and immensely important."

— U. Manber, *Introduction to Algorithms*

Um vetor de inteiros $v[0..n-1]$ é **crescente** se $v[0] \le v[1] \le \cdots \le v[n-1]$ e **decrescente** se $v[0] \ge v[1] \ge \cdots \ge v[n-1]$. O vetor é **ordenado** se for crescente ou decrescente.

Este capítulo estuda o problema de encontrar um dado inteiro em um dado vetor ordenado. Mais precisamente, dado um inteiro x e um vetor crescente $v[0..n-1]$, queremos encontrar um índice m tal que $v[m] = x$.

7.1 O problema

Comecemos com uma decisão de projeto. Em lugar de perguntar onde x está no vetor $v[0..n-1]$, é mais útil e mais conveniente perguntar onde x *deveria* estar. Nosso problema pode ser formulado assim: *dado um inteiro x e um vetor crescente $v[0..n-1]$, encontrar um índice j tal que*

$$v[j-1] < x \le v[j]\,. \tag{7.1}$$

De posse de um tal j, é muito fácil resolver o problema enunciado na introdução do capítulo: basta comparar x com $v[j]$.

Qualquer valor de j no intervalo fechado $0..n$ pode ser solução do problema. Nos dois extremos do intervalo, 0 e n, a condição (7.1) deve ser interpretada com inteligência: se $j = 0$ então a condição se reduz a $x \le v[0]$, pois $v[-1]$ não faz sentido; se $j = n$, a condição se reduz a $v[n-1] < x$, pois $v[n]$ não faz sentido. Tudo se passa como se nosso vetor tivesse um componente imaginário $v[-1]$ com valor $-\infty$ e um componente imaginário $v[n]$ com valor $+\infty$.

Precisamos tomar mais uma decisão de projeto. Qual o menor valor de n que devemos aceitar? Embora o problema faça sentido quando n vale 0 (a solução do problema é 0 nesse caso), suporemos sempre que

$$n \geq 1\,,$$

pois isso simplifica um pouco o raciocínio.

0												$n-1$
111	222	333	444	555	555	666	777	888	888	888	999	999

Figura 7.1: Um vetor crescente $v[0\,..\,n-1]$, com $n = 13$. Queremos encontrar j tal que $v[j-1] < x \leq v[j]$. Se x vale 555 então o valor correto de j é 4. Se x vale 1000, o valor correto de j é 13. Se x vale 110 ou 111, o valor correto de j é 0.

7.2 Busca sequencial

Comecemos com um algoritmo óbvio e simples (mas lento) conhecido como *busca sequencial*:

```
int BuscaSequencial (int x, int n, int v[]) {
   int j = 0;
   while (j < n && v[j] < x) ++j;
   return j;
}
```

O consumo de tempo do processo iterativo comandado pelo `while` é proporcional ao número de iterações, e este número não passa de n. O consumo de tempo das demais linhas do código pode ser ignorado pois não depende de n. Podemos dizer, portanto, que o consumo de tempo da função é

$$\text{proporcional a } n$$

no pior caso. Em outras palavras, a função não consome mais que n unidades de tempo.[1] Suponha, por exemplo, que a função consome 1 milissegundo, no

[1] A unidade de tempo depende do computador e dos detalhes da implementação da função, mas não do valor de n.

pior caso, para um determinado valor de n. Se tivermos $1000\,n$ no lugar de n, a função consumirá 1000 milissegundos no pior caso.

O algoritmo de busca sequencial é ineficiente porque, no pior caso, compara x com cada um dos elementos do vetor. A próxima seção mostra que é possível fazer algo muito melhor.

Exercícios

7.2.1 Critique a seguinte formulação do problema de busca: dado x e um vetor crescente $v[0\,..\,n{-}1]$, encontrar um índice j tal que $v[j{-}1] \le x \le v[j]$. Critique a formulação baseada em "$v[j{-}1] < x < v[j]$".

7.2.2 Invariante. Na função `BuscaSequencial`, qual o invariante do processo iterativo controlado pelo `while`?

7.2.3 Critique a seguinte versão da função `BuscaSequencial`:

```
int j = 0;
while (v[j] < x && j < n) ++j;
return j;
```

7.2.4 Versão recursiva. Escreva uma versão recursiva da função `BuscaSequencial`.

7.3 Busca binária

A busca binária é muito mais eficiente que a busca sequencial. Ela se baseia no método que usamos às vezes para encontrar uma palavra num dicionário.

```
/* Esta função recebe um vetor crescente v[0..n-1] com
 * n >= 1 e um inteiro x. Ela devolve um índice j
 * em 0..n tal que v[j-1] < x <= v[j]. */
int BuscaBinária (int x, int n, int v[]) {
   int e, m, d;
   e = -1; d = n;
   while (e < d-1) {
      m = (e + d)/2;
      if (v[m] < x) e = m;
      else d = m;
   }
   return d;
}
```

(Os nomes das variáveis não foram escolhidos ao acaso: e lembra "esquerda", m lembra "meio" e d lembra "direita".) O resultado da divisão por `2` na expressão `(e+d)/2` é automaticamente truncado pois só envolve variáveis e constantes do tipo `int`. Portanto, o valor da expressão é $\lfloor \frac{e+d}{2} \rfloor$.

Exercícios

7.3.1 Discuta e critique a elegância da seguinte variante da função `BuscaBinária`:

```
int e, m, d;
if (v[n-1] < x) return n;
if (x <= v[0]) return 0;
e = 0; d = n-1;
while (e < d-1) {
   m = (e + d)/2;
   if (v[m] < x) e = m;
   else d = m; }
return d;
```

7.3.2 Suponha que $v[i] = i$ para todo i. Execute a função `BuscaBinária` com $n = 9$ e $x = 3$. Repita o exercício com $n = 14$ e $x = 7$. Repita o exercício com $n = 15$ e $x = 7$.

7.3.3 Execute a função `BuscaBinária` com $n = 16$. Quais os possíveis valores de m na primeira iteração? Quais os possíveis valores de m na segunda iteração? Na terceira? Na quarta?

7.3.4 Na função `BuscaBinária`, verifique que m pertence ao intervalo $0..n-1$ (e portanto $v[m]$ faz sentido) sempre que o fragmento de código "`if (`$v[m]$ `<` x`)`" é executado.

7.3.5 Confira a validade da seguinte afirmação: quando $n+1$ é uma potência de 2, o valor da expressão `(`e`+`d`)` é divisível por 2 em todas as iterações da função `BuscaBinária` (quaisquer que sejam v e x).

7.3.6 Responda as seguintes perguntas sobre a função `BuscaBinária`. Que acontece se "`while (`e `<` d`-1)`" for substituído por "`while (`e `<` d`)`"? Que acontece se "`if (`$v[m]$ `<` x`)`" for substituído por "`if (`$v[m]$ `<=` x`)`"? Que acontece se "e `=` m" for substituído por "e `=` m`+1`" ou por "e `=` m`-1`"? Que acontece se "d `=` m" for substituído por "d `=` m`+1`" ou por "d `=` m`-1`"?

7.4 Prova da correção do algoritmo

Para compreender a função `BuscaBinária`, basta verificar o seguinte invariante: no início de cada repetição do `while`, imediatamente antes da comparação de e com d`-1`, vale a relação

$$v[e] < x \leq v[d] \tag{7.2}$$

(veja Exercício 7.4.1 abaixo). O algoritmo foi, na verdade, construído a partir desta relação.

Figura 7.2: Início de uma iteração da função `BuscaBinária`.

No início da primeira iteração, a relação (7.2) está automaticamente satisfeita, pois $v[-1]$ e $v[n]$ não fazem sentido. (Se preferir, você pode imaginar que $v[-1] = -\infty$ e $v[n] = +\infty$. Mas o código da função não comete o erro de usar $v[-1]$ ou $v[n]$.)

No início da última iteração temos $e = d - 1$ (veja Exercício 7.4.2 abaixo) e portanto o invariante (7.2) se reduz a $v[d-1] < x \leq v[d]$, donde d é a solução de nosso problema. Assim, ao devolver d, o algoritmo está cumprindo o que prometeu fazer.

Resta verificar que a execução do algoritmo termina. No início de cada iteração, o número de elementos do vetor em jogo é $d - e - 1$. Como $e < m < d$ (veja Exercício 7.4.3), tanto $d - m - 1$ quanto $m - e - 1$ são estritamente menores que $d - e - 1$. Portanto, o tamanho do vetor em jogo diminui a cada iteração e a execução do algoritmo para, mais cedo ou mais tarde.

Exercícios

7.4.1 Suponha que estamos no início de uma iteração (que não a última) da função `BuscaBinária`. Suponha que vale a relação (7.2). Mostre que (7.2) vale no início da próxima iteração.

7.4.2 Mostre que no início da última iteração da função `BuscaBinária` temos $e = d - 1$.

7.4.3 Na função `BuscaBinária`, mostre que temos $e < m < d$ imediatamente depois da atribuição "m `= (e+d)/2`".

7.5 Desempenho do algoritmo

Quantas iterações a função `BuscaBinária` executa? Em cada iteração, o tamanho do vetor em jogo é $d - e - 1$. No início da primeira iteração, o tamanho do vetor é n. No início da segunda, o tamanho é aproximadamente $n/2$. No

início da terceira, aproximadamente $n/4$. No início da $(k{+}1)$-ésima, aproximadamente $n/2^k$. Quando $k > \log_2 n$, temos $n/2^k < 1$ e a execução do algoritmo para. Assim, o número de iterações é (veja Exercício 1.2.4) aproximadamente

$$\log_2 n\,.$$

O consumo de tempo da função é proporcional ao número de iterações e portanto proporcional a $\log_2 n$. Esse consumo cresce com n muito mais devagar que o consumo da busca sequencial, pois log transforma multiplicações em somas. Por exemplo, se cada iteração consome 1 milissegundo, uma busca em n elementos consome $\log_2 n$ milissegundos, uma busca em $2n$ elementos consome apenas $1 + \log_2 n$ milissegundos, uma busca em $4n$ elementos consome só $2 + \log_2 n$ milissegundos e uma busca em $1024\,n$ elementos consumirá tão somente $10 + \log_2 n$ milissegundos.

Exercícios

7.5.1 Faça uma tabela de valores $\lfloor \log_2 n \rfloor$ para $n = 10, 10^2, 10^3, 10^4, 10^5$. (Veja o Exercício 1.2.4.)

7.5.2 Se t segundos são necessários para fazer uma busca binária em um vetor com n elementos, quantos segundos serão necessários para fazer uma busca em n^2 elementos?

7.5.3 OVERFLOW ARITMÉTICO. Se o número de elementos do vetor $v[0\,..\,n{-}1]$ estiver próximo de `INT_MAX` (veja Seção K.5), o código da função `BuscaBinária` pode descarrilar ao calcular o valor da expressão `(e+d)/2`, em virtude de um *overflow* aritmético. Como evitar isso?

7.6 Exercícios: variantes do código

Há muitas maneiras de escrever o código da busca binária. Todas exigem cuidado e atenção aos detalhes, pois é muito fácil escrever uma versão que dá respostas erradas ou "entra em *loop*". Os exercícios abaixo introduzem algumas versões diferentes da discutida na Seção 7.3. Todas prometem devolver um índice j no intervalo $0\,..\,n$ tal que $v[j-1] < x \le v[j]$.

7.6.1 Mostre que a seguinte variante da função `BuscaBinária` funciona corretamente.

```
e = 0; d = n;
while (e < d) { /* v[e-1] < x <= v[d] */
   m = (e + d)/2;
   if (v[m] < x) e = m + 1;
   else d = m;
} /* e == d */
return d;
```

(Esta versão é quase tão elegante quanto a versão discutida na Seção 7.3.) Que acontece se trocarmos "`while (e < d)`" por "`while (e <= d)`"? Que acontece se trocarmos "`(e+d)/2`" por "`(e-1+d)/2`"?

7.6.2 Mostre que a seguinte versão da função `BuscaBinária` funciona corretamente. Ela é um pouco menos elegante que as versões anteriores.

```
e = 0; d = n-1;
while (e <= d) { /* v[e-1] < x <= v[d+1] */
   m = (e + d)/2;
   if (v[m] < x) e = m + 1;
   else d = m-1;
} /* e == d + 1 */
return d+1;
```

7.6.3 A seguinte alternativa para a função `BuscaBinária` funciona corretamente? Que acontece se trocarmos "`(e+d)/2`" por "`(e+d+1)/2`"?

```
e = -1; d = n-1;
while (e < d) {
   m = (e + d)/2;
   if (v[m] < x) e = m;
   else d = m - 1; }
return d + 1;
```

7.7 Versão recursiva da busca binária

A formulação do problema que usamos até aqui não se presta, diretamente, a uma solução recursiva. Será necessário reformular o problema ligeiramente. Para fazer a transição da formulação anterior para a nova usaremos a seguinte função-embalagem, que tem a mesma documentação que `BuscaBinária`:

```
int BuscaBinária2 (int x, int n, int v[]) {
   return BuscaBinR (x, -1, n, v);
}
```

A função recursiva `BuscaBinR` procura x no vetor crescente v`[`e`+1..`d`-1]` supondo que o valor de x está entre os extremos v`[`e`]` e v`[`d`]`:

```
/* O vetor v[e+1..d-1] é crescente e o inteiro x é tal que
 * v[e] < x <= v[d]. A função devolve um índice j no
 * intervalo e+1..d tal que v[j-1] < x <= v[j]. */
```

```
int BuscaBinR (int x, int e, int d, int v[]) {
   if (e == d-1) return d;
   else {
      int m = (e + d)/2;
      if (v[m] < x)
         return BuscaBinR (x, m, d, v);
      else
         return BuscaBinR (x, e, m, v);
   }
}
```

Quando a função `BuscaBinR` é invocada com argumentos $(x, -1, n, v)$, ela invoca a si mesma cerca de $\lfloor \log_2 n \rfloor$ vezes. Este número de invocações é a "profundidade" da recursão.

Exercícios

7.7.1 Discute a seguinte variante da função `BuscaBinária2`:

```
if (v[n-1] < x) return n;
if (x <= v[0]) return 0;
return BuscaBinR (x, 0, n-1, v);
```

7.7.2 Mostre que as condições descritas na documentação da função `BuscaBinR` estão satisfeitas no momento em que `BuscaBinR` é invocada por `BuscaBinária2`.

7.7.3 Considere a função `BuscaBinR`. Suponha que v`[m]` $< x$. Verifique que v`[m]` $< x \leq v$`[`d`]`, mostrando assim que a função `BuscaBinR` pode ser invocada com argumentos x, `m`, d, v. Agora suponha que v`[m]` $\geq x$ e verifique que v`[`e`]` $< x \leq v$`[m]`, mostrando assim que `BuscaBinR` pode ser invocada com argumentos x, e, `m`, v.

7.7.4 Leia o verbete *Binary search algorithm* na Wikipedia [21].

7.8 Exercícios: variações sobre o tema

O algoritmo de busca binária é um verdadeiro "ovo de Colombo". A ideia básica do algoritmo é o ponto de partida de muitos algoritmos eficientes. Os exercícios abaixo procuram explorar o tema.

7.8.1 Outra formulação da busca binária. Escreva uma versão da busca binária que receba um inteiro x e um vetor $v[0\,..\,n{-}1]$ e devolva j tal que em $v[j{-}1] \leq x < v[j]$ (note a posição de "$\leq$" e "$<$"). Quais os possíveis valores de j?

7.8.2 Vetor decrescente. Escreva uma versão da busca binária para resolver o

seguinte problema: dado um inteiro x e um vetor decrescente $v[0\,..\,n{-}1]$, encontrar j tal que $v[j{-}1] > x \geq v[j]$.

7.8.3 Busca simplificada. Escreva uma função que resolva o problema formulado na introdução do capítulo: ao receber um inteiro x e um vetor crescente $v[0\,..\,n{-}1]$, devolva um índice m tal que $v[m] = x$ ou devolva -1 se tal m não existe. Escreva duas versões: uma iterativa e uma recursiva.

7.8.4 Vetor de strings. Suponha que cada elemento do vetor $v[0\,..\,n{-}1]$ é uma string. Suponha também que o vetor está em ordem lexicográfica (veja Seção G.3). Escreva uma função que receba uma string x e devolva um índice j tal que x é igual a $v[j]$. Se tal j não existe, a função deve devolver -1.

7.8.5 Vetor de structs. Suponha que cada elemento do vetor $v[0\,..\,n{-}1]$ é uma struct (veja Apêndice E) com dois campos: o nome de um aluno e o número do aluno. Suponha que o vetor está em ordem crescente de números. Escreva uma função de busca binária que receba o número de um aluno e devolva o seu nome. Se o número não estiver no vetor, a função deve devolver a string vazia.

7.8.6 Procurando por $v[i] = i$. Escreva uma função que receba um vetor estritamente crescente[2] $v[0\,..\,n{-}1]$ de números inteiros e devolva um índice i entre 0 e $n{-}1$ tal que $v[i] = i$; se tal i não existe, a função deve devolver -1. O seu algoritmo não deve fazer mais que $\lfloor \log_2 n \rfloor$ comparações envolvendo elementos de v.

7.8.7 Escreva uma função eficiente que receba inteiros positivos k e n e calcule k^n. Quantas multiplicações sua função executa? (Compare com o Exercício 2.3.10.)

7.8.8 A seguinte função recursiva pretende encontrar o valor de um elemento máximo do vetor $v[e\,..\,d]$ supondo $e \leq d$. O vetor não está necessariamente ordenado. A função está correta? Ela é mais rápida que a correspondente versão iterativa? Qual a profundidade da recursão?

```
int max (int e, int d, int v[]) {
   if (e == d) return v[d];
   else {
      int m, maxe, maxd;
      m = (e + d)/2;
      maxe = max (e, m, v);
      maxd = max (m + 1, d, v);
      if (maxe >= maxd) return maxe;
      else return maxd; } }
```

[2] Um vetor $v[0\,..\,n{-}1]$ é **estritamente crescente** se $v[0] < v[1] < \cdots < v[n{-}1]$.

Capítulo 8

Ordenação: algoritmos elementares

Colocar um vetor numérico em ordem crescente é o primeiro passo na solução de muitos problemas práticos. Um vetor pode ser ordenado de muitas maneiras diferentes: algumas elementares, outras mais sofisticadas e eficientes. Assim, o problema da ordenação é um verdadeiro laboratório de projeto de algoritmos. Trataremos do assunto neste capítulo e nos três capítulos seguintes.

8.1 O problema da ordenação

Um vetor $v[0\,..\,n{-}1]$ é **crescente** se $v[0] \le v[1] \le \cdots \le v[n{-}1]$. O problema da ordenação de um vetor consiste no seguinte:

> Rearranjar (ou seja, permutar) os elementos de um vetor $v[0\,..\,n{-}1]$ de tal modo que ele se torne crescente.

Este capítulo discute dois algoritmos simples para o problema. Os três capítulos seguintes examinam algoritmos mais sofisticados e eficientes.

Exercícios

8.1.1 Escreva uma função que verifique se um dado vetor $v[0\,..\,n{-}1]$ é crescente.

8.1.2 Leia o verbete *Sorting algorithm* na Wikipedia [21].

8.2 Algoritmo de inserção

O algoritmo de ordenação por inserção é muito popular; ele é frequentemente usado para colocar em ordem um baralho de cartas.

```
/* Esta função rearranja o vetor v[0..n-1] em ordem
 * crescente. */
void Inserção (int n, int v[]) {
   int i, j, x;
   for (j = 1; /*A*/ j < n; j++) {
      x = v[j];
      for (i = j-1; i >= 0 && v[i] > x; i--)
         v[i+1] = v[i];
      v[i+1] = x;
   }
}
```

Para entender o algoritmo, basta observar que no início de cada repetição do `for` externo, ou seja, a cada passagem pelo ponto `A`,

1. o vetor $v[0\,..\,n{-}1]$ é uma permutação do vetor original e
2. o vetor $v[0\,..\,j{-}1]$ é crescente.

Estas propriedades invariantes são trivialmente verdadeiras no início da primeira iteração, quando j vale 1, e permanecem verdadeiras no início das iterações subsequentes. No início da última iteração, j vale n e portanto o vetor $v[0\,..\,n{-}1]$ está na ordem desejada. (Note que a última iteração é interrompida logo no início, pois a condição $j < n$ é falsa.)

0	crescente		$j{-}1$	j						$n{-}1$
444	555	555	666	777	222	999	222	999	222	999

Figura 8.1: Vetor $v[0\,..\,n{-}1]$ no início de uma iteração da função `Inserção`.

Desempenho do algoritmo. O consumo de tempo da função `Inserção` é proporcional ao número de execuções da comparação "`v[i] > x`". Calculemos esse número. Para cada valor de j, a variável i assume no máximo j valores, a saber, $j-1, j-2, \ldots, 0$. Como j varia de 1 a n, o número de execuções da

comparação "`v[i] > x`" é igual a $\sum_{j=1}^{n-1} j$ no pior caso. A soma vale $n\,(n-1)/2$ e este número é essencialmente igual a $n^2/2$ quando n é grande. Pode-se dizer, portanto, que o consumo de tempo da função é

$$\text{proporcional a } n^2$$

no pior caso. Em outras palavras, a função consome no máximo n^2 unidades de tempo.[1] Se a ordenação de n números consumir t milissegundos, a ordenação de $2n$ números consumirá $4t$ milissegundos e a ordenação de $10n$ números consumirá $100t$ milissegundos. Portanto, o algoritmo é lento quando n é grande.

Exercícios

8.2.1 No código da função `Inserção`, troque "`v[i] > x`" por "`v[i] >= x`". A nova função continua produzindo uma ordenação crescente de `v[0..n-1]`?

8.2.2 No código da função `Inserção`, que acontece se trocarmos "`for (j = 1`" por "`for (j = 0`"? Que acontece se trocarmos "`v[i+1] = x`" por "`v[i] = x`"?

8.2.3 Critique a seguinte implementação do algoritmo de ordenação por inserção:

```
int i, j, x;
for (j = 1; j < n; j++) {
   for (i = j-1; i >= 0 && v[i] > v[i+1]; i--) {
      x = v[i]; v[i] = v[i+1]; v[i+1] = x; } }
```

8.2.4 Critique a seguinte implementação do algoritmo de ordenação por inserção:

```
int h, i, j, x;
for (j = 1; j < n; j++) {
   x = v[j];
   for (h = 0; h < j && v[h] <= x; h++) ;
   for (i = j-1; i >= h; i--) v[i+1] = v[i];
   v[h] = x; }
```

8.2.5 Escreva uma versão do algoritmo de inserção que tenha o seguinte invariante: no início de cada iteração, o vetor $v[j{+}1\,..\,n{-}1]$ é crescente.

8.2.6 Busca binária. O `for` interno na função `Inserção` tem a missão de encontrar o ponto onde `v[j]` deve ser inserido em `v[0..j-1]`, ou seja, encontrar o índice i tal que `v[i]` $\le$ `v[j]` $<$ `v[i+1]`. Considere fazer isso com uma busca binária (veja Seção 7.3). Analise o resultado.

8.2.7 Ordem estritamente crescente. Escreva uma função que rearranje um vetor $v[0\,..\,n{-}1]$ de modo que ele fique em ordem estritamente crescente.

8.2.8 Ordem decrescente. Escreva uma função que permute os elementos de um vetor $v[0\,..\,n{-}1]$ de modo que eles fiquem em ordem decrescente.

[1] A unidade de tempo depende do computador e dos detalhes da implementação da função, mas não do valor de n.

8.2.9 Versão recursiva. Escreva uma versão recursiva do algoritmo de ordenação por inserção.

8.2.10 Animações. Veja animações do algoritmo de inserção nas páginas de Harrison [8] e Morin [13] da teia www.

8.2.11 Leia o verbete *Insertion sort* na Wikipedia [21].

8.3 Algoritmo de seleção

O algoritmo de ordenação por seleção é baseado na ideia de escolher o menor elemento do vetor, depois o segundo menor,[2] e assim por diante.

```
/* Rearranja o vetor v[0..n-1] em ordem crescente. */
void Seleção (int n, int v[]) {
   int i, j, min, x;
   for (i = 0; /*A*/ i < n-1; i++) {
      min = i;
      for (j = i+1; j < n; j++)
         if (v[j] < v[min]) min = j;
      x = v[i]; v[i] = v[min]; v[min] = x;
   }
}
```

Para entender como e por que o algoritmo funciona, basta observar que no início de cada repetição do `for` externo, ou seja, a cada passagem pelo ponto `A`, valem os seguintes invariantes:

1. $v[0\,..\,n{-}1]$ é uma permutação do vetor original,
2. $v[0\,..\,i{-}1]$ está em ordem crescente e
3. $v[i-1] \le v[j]$ para $j = i, i{+}1, \ldots, n{-}1$.

A tradução do invariante 3 para linguagem humana é a seguinte: $v[0\,..\,i{-}1]$ contém todos os elementos "pequenos" do vetor original e $v[i\,..\,n{-}1]$ contém todos os elementos "grandes". Os três invariantes garantem que no início de cada iteração os elementos $v[0], \ldots, v[i-1]$ já estão em suas posições definitivas.

Desempenho do algoritmo. Uma análise semelhante à que fizemos para o algoritmo de inserção mostra que o algoritmo de seleção faz cerca de $n^2/2$

[2] A rigor, deveríamos dizer "selecionar *um* menor elemento, depois *um* segundo menor".

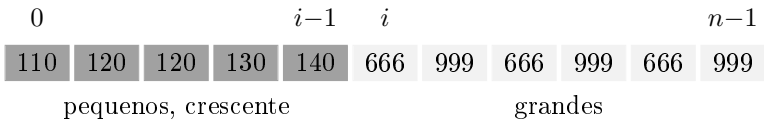


Figura 8.2: Vetor $v[0\,..\,n{-}1]$ no início de uma iteração da função `Seleção`.

comparações entre elementos do vetor. Portanto, consome n^2 unidades de tempo no pior caso.

Exercícios

8.3.1 Que acontece se trocarmos "`for (`i `= 0`" por "`for (`i `= 1`" no código da função `Seleção`? Que acontece se trocarmos "`for (`i `= 0; `i `< `n`-1`" por "`for (`i `= 0; `i `< `n"?

8.3.2 Troque "v`[`j`] < `v`[min]`" por "v`[`j`] <= `v`[min]`" no código de `Seleção`. A nova função continua produzindo uma ordenação crescente de v`[0..`n`-1]`?

8.3.3 Ordem decrescente. Escreva uma função que permute os elementos de um vetor $v[0\,..\,n{-}1]$ de modo que eles fiquem em ordem decrescente.

8.3.4 Versão recursiva. Escreva uma versão recursiva do algoritmo de ordenação por seleção.

8.3.5 Animações. Veja animações do algoritmo de seleção nas páginas de Harrison [8] e Morin [13].

8.3.6 Leia o verbete *Selection sort* na Wikipedia [21].

8.4 Exercícios: ordenação de strings e listas

8.4.1 Ordenação de strings. Escreva uma função que coloque um vetor de strings em ordem lexicográfica (veja Seção G.3). Faça duas versões: uma baseada no algoritmo de inserção e outra baseada no algoritmo de seleção.

8.4.2 Ordenação de arquivo. Escreva uma função que rearranje as linhas de um arquivo (veja Apêndice H) em ordem lexicográfica (veja Seção G.3). Compare com o utilitário `sort` presente em todo sistema UNIX e GNU/Linux.

8.4.3 Ordenação de structs. Suponha que cada elemento de um vetor é um registro que consiste em um inteiro e uma string:

```
struct elem {int i; char *s;};
```

Escreva uma função que rearranje o vetor de modo que os campos `i` fiquem em ordem crescente. Escreva outra função que rearranje o vetor de modo que os campos `s` fiquem em ordem lexicográfica (veja Seção G.3).

8.4.4 Ordenação de lista encadeada. Escreva uma função que ordene uma lista encadeada. Inspire-se no algoritmo de ordenação por inserção. Faça duas versões: uma para lista com cabeça e outra para lista sem cabeça. (Sua função precisa devolver alguma coisa?). Repita o exercício com base no algoritmo de ordenação por seleção.

8.4.5 Projeto de programação. Digamos que duas palavras são *equivalentes* se uma é anagrama da outra, ou seja, se a sequência de letras de uma é permutação da sequência de letras da outra. Por exemplo, "`aberto`" e "`rebato`" são equivalentes.

Uma *classe de equivalência* de palavras é um conjunto de palavras duas a duas equivalentes. Escreva um programa que receba um arquivo de palavras (uma palavra por linha) e extraia desse arquivo uma classe de equivalência de tamanho máximo. Aplique o seu programa ao arquivo de palavras `www.ime.usp.br/~pf/algoritmos/dicios/`, que contém todas as palavras do português falado no Brasil (o arquivo foi extraído do Dicionário br.ispell [20]).

8.5 Ordenação estável

Um algoritmo de ordenação é **estável** se não altera a posição relativa de elementos que têm um mesmo valor. Por exemplo, se o vetor tiver dois elementos de valor 222, um algoritmo de ordenação estável manterá o primeiro 222 antes do segundo.

vetor original:	444	555	666	777	333	222_1	111	222_2	888
vetor ordenado:	111	222_1	222_2	333	444	555	666	777	888

Figura 8.3: Ordenação estável. O vetor original tem dois elementos com valor 222 (índices $_1$ e $_2$ são usados para distinguir o primeiro do segundo). No vetor ordenado, o primeiro destes elementos continua à frente do segundo.

Suponha, por exemplo, que os elementos de um vetor são pares da forma (d, m) que representam datas de um certo ano: a primeira componente representa o dia e a segunda representa o mês. (Compare com o Exercício 8.4.3.) Suponha que o vetor está em ordem crescente das componentes d:

$$(1,12),\ (7,12),\ (16,3),\ (25,9),\ (30,3),\ (30,6),\ (31,3).$$

Agora ordene o vetor pelas componentes m. Se usarmos um algoritmo de ordenação estável, o resultado estará em ordem cronológica:

$$(16,3),\ (30,3),\ (31,3),\ (30,6),\ (25,9),\ (1,12),\ (7,12).$$

Se o algoritmo de ordenação não for estável, o resultado pode não ficar em ordem cronológica:

$$(30,3),\ (16,3),\ (31,3),\ (30,6),\ (25,9),\ (7,12),\ (1,12).$$

Exercícios

8.5.1 O algoritmo de ordenação por inserção (Seção 8.2) é estável?

8.5.2 Na código da função `Inserção` (Seção 8.2), troque a comparação "v`[`i`] > x`" por "v`[`i`] >= x`". A nova função faz uma ordenação estável de v`[0..`n`-1]`?

8.5.3 O algoritmo de ordenação por seleção (Seção 8.3) é estável?

Capítulo 9

Ordenação: algoritmo Mergesort

Considere o problema da ordenação enunciado na introdução na Seção 8.1: *permutar os elementos de um vetor $v[0\,..\,n{-}1]$ de modo que ele se torne crescente*. O Capítulo 8 examinou dois algoritmos simples para o problema. Este capítulo examina um algoritmo mais sofisticado e mais rápido baseado na estratégia "dividir para conquistar".

9.1 Intercalação de vetores ordenados

Antes de tratar do problema da ordenação propriamente dito, é preciso resolver um problema auxiliar: *dados vetores crescentes $v[p\,..\,q{-}1]$ e $v[q\,..\,r{-}1]$, rearranjar $v[p\,..\,r{-}1]$ em ordem crescente*. Podemos dizer que o problema consiste em "intercalar" os dois vetores dados.

É fácil resolver o problema em tempo proporcional ao quadrado de $r - p$: basta aplicar um dos algoritmos do Capítulo 8 ao vetor $v[p\,..\,r{-}1]$ ignorando o fato de que as duas "metades" estão ordenadas. Mas é possível resolver o problema de maneira bem mais eficiente. Para isso, será preciso usar um vetor auxiliar, digamos w, do mesmo tipo e mesmo tamanho que $v[p\,..\,r{-}1]$.

p						$q{-}1$	q			$r{-}1$
111	333	555	555	777	999	999	222	444	777	888

Figura 9.1: Rearranjar o vetor $v[p\,..\,r{-}1]$ em ordem crescente sabendo que $v[p\,..\,q{-}1]$ e $v[q\,..\,r{-}1]$ já estão em ordem crescente.

```
/* A função recebe vetores crescentes v[p..q-1] e v[q..r-1]
 * e rearranja v[p..r-1] em ordem crescente. */
void Intercala (int p, int q, int r, int v[]) {
   int i, j, k, *w;
   w = malloc ((r-p) * sizeof (int));
   i = p; j = q; k = 0;
   while (i < q && j < r) {
      if (v[i] <= v[j]) w[k++] = v[i++];
      else w[k++] = v[j++];
   }
   while (i < q) w[k++] = v[i++];
   while (j < r) w[k++] = v[j++];
   for (i = p; i < r; i++) v[i] = w[i-p];
   free (w);
}
```
[1]

Desempenho da intercalação. A função `Intercala` consome tempo proporcional ao número de comparações entre elementos do vetor. Esse número é menor que $r - p$. Podemos dizer, então, que o consumo de tempo da função no pior caso é

proporcional ao número de elementos do vetor.

O algoritmo de intercalação é, portanto, muito eficiente.

Exercícios

9.1.1 A função `Intercala` está correta nos casos extremos $p = q$ e $q = r$?

9.1.2 Que acontece se trocarmos "`else` w`[k++] =` v`[j++]`" por "`if (`v`[i] >` v`[j]) `w`[k++] =` v`[j++]`" no código da função `Intercala`?

9.1.3 Na função `Intercala`, troque o par de linhas "`while (j <` r`) ... ; for ... =` w`[i-`p`];`" por "`for (i =` p`; i < j; i++)` v`[i] =` w`[i-`p`];`". Discuta o resultado.

9.1.4 Discuta a seguinte alternativa para a função `Intercala`:

```
i = p; j = q;
for (k = 0; k < r-p; k++) {
   if (j >= r || (i < q && v[i] <= v[j]))  w[k] = v[i++];
   else  w[k] = v[j++]; }
for (i = p; i < r; i++)  v[i] = w[i-p];
```

[1] Veja Seção F.3.

9.1.5 Critique a seguinte alternativa para a função `Intercala`:

```
i = p; j = q; k = 0;
while (k < r-p) {
   while (i < q && v[i] <= v[j])
      w[k++] = v[i++];
   while (j < r && v[j] <= v[i])
      w[k++] = v[j++]; }
for (i = p; i < r; i++)  v[i] = w[i-p];
```

9.1.6 A seguinte alternativa para a função `Intercala` não usa vetor auxiliar. Ela está correta? Quais os invariantes do `while`? Qual o consumo de tempo?

```
int i = p, k, t;
while (i < q && q < r) {
   if (v[i] >= v[q]) {
      t = v[q];
      for (k = q-1; k >= i; k--) v[k+1] = v[k];
      v[i] = t;
      q++; }
   i++; }
```

9.1.7 Versão recursiva. Escreva uma versão recursiva do algoritmo da intercalação. Antes, convém reformular o problema da seguinte maneira: dados vetores crescentes $u[0\,..\,m{-}1]$ e $v[0\,..\,n{-}1]$, produzir um vetor crescente $w[0\,..\,m{+}n{-}1]$ que contenha o resultado da intercalação dos dois vetores.

9.1.8 Estabilidade. Um algoritmo de intercalação é *estável* (veja Seção 8.5) se não altera a posição relativa de elementos de mesmo valor. A função `Intercala` é estável? Se "`v[i] <= v[j]`" for trocada por "`v[i] < v[j]`", a função fica estável?

9.1.9 Intercalação com sentinelas. Sedgewick [18] escreve a função `Intercala` de maneira muito interessante. Eis um esboço do código:

```
int w[MAX], i, j, k;
for (i = p; i < q; i++) w[i] = v[i];
for (j = q; j < r; j++) w[r+q-j-1] = v[j];
i = p; j = r-1;
for (k = p; k < r; k++)
   if (w[i] < w[j]) v[k] = w[i++];
   else v[k] = w[j--];
```

Este esboço está sujeito à restrição `r` $\leq$ `MAX` e desperdiça espaço se `p` for maior que 0. Reescreva o código depois de corrigir estes defeitos.

9.1.10 Listas encadeadas. Chame de lecr qualquer lista encadeada sem cabeça que contém uma sequência crescente de números inteiros. Escreva uma função que intercale duas lecr dadas, produzindo assim uma terceira lecr. Sua função não deve alocar novas células na memória, mas reaproveitar as células das duas listas dadas.

9.2 O algoritmo Mergesort

Agora podemos usar a função `Intercala` para escrever um algoritmo rápido de ordenação. Nosso código é recursivo. A base da recursão é o caso $p \geq r - 1$; nesse caso não é preciso fazer nada.

```
/* Esta função rearranja o vetor v[p..r-1] em ordem
 * crescente. */
void Mergesort (int p, int r, int v[]) {
   if (p < r-1) {
      int q = (p + r)/2;
      Mergesort (p, q, v);
      Mergesort (q, r, v);
      Intercala (p, q, r, v);
   }
}
```

(O resultado da divisão por 2 na expressão `(p+r)/2` é automaticamente truncado pois só envolve variáveis e constantes do tipo `int`. Portanto, o valor da expressão é $\lfloor \frac{p+r}{2} \rfloor$.) Para rearranjar `v[0..n-1]` em ordem crescente basta dizer `Mergesort (0, n, v)`.

0	1	2	3	4	5	6	7	8	9	10
999	111	222	999	888	333	444	777	555	666	555
999	111	222	999	888	333	444	777	555	666	555
999	111	222	999	888	333	444	777	555	666	555
⋮										
111	999	222	888	999	333	444	777	555	555	666
111	222	888	999	999	333	444	555	555	666	777
111	222	333	444	555	555	666	777	888	999	999

Figura 9.2: Algoritmo Mergesort aplicado ao vetor $v[0..10]$. Nas primeiras "rodadas", o algoritmo não faz mais que quebrar o vetor em segmentos (veja as gradações de cinza). Nas rodadas subsequentes, segmentos vizinhos são intercalados.

Exercícios

9.2.1 Que acontece se trocarmos "(p+r)/2" por "(p+r-1)/2" no código de `Mergesort`? Que acontece se trocarmos "(p+r)/2" por "(p+r+1)/2"?

9.2.2 Submeta um vetor v[1..4] à função `Mergesort`. Teremos a seguinte sequência de invocações da função:

```
Mergesort (1, 5, v)
   Mergesort (1, 3, v)
      Mergesort (1, 2, v)
      Mergesort (2, 3, v)
   Mergesort (3, 5, v)
      Mergesort (3, 4, v)
      Mergesort (4, 5, v)
```

Faça uma figura análoga para um vetor v[1..5].

9.2.3 A função `Mergesort` é estável? (Veja Seção 8.5 e Exercício 9.1.8.)

9.2.4 Discuta a seguinte implementação da função `Mergesort`:

```
if (p < r) {
   int q = (p + r)/2;
   Mergesort (p, q, v);
   Mergesort (q, r, v);
   Intercala (p, q, r, v); }
```

9.2.5 Discuta a seguinte implementação da função `Mergesort`:

```
if (p < r-1) {
   int q = (p + r - 1)/2;
   Mergesort (p, q, v);
   Mergesort (q, r, v);
   Intercala (p, q, r, v); }
```

9.2.6 Critique a implementação da função `Mergesort` abaixo. Repita o exercício com "(p+r+1)/2" no lugar de "(p+r)/2".

```
if (p < r-1) {
   int q = (p + r)/2;
   Mergesort (p, q-1, v);
   Mergesort (q-1, r, v);
   Intercala (p, q-1, r, v); }
```

9.2.7 Critique a seguinte implementação da função `Mergesort`:

```
if (p < r-1) {
   q = r - 1;
   Mergesort (p, q, v);
   Intercala (p, q, r, v); }
```

9.2.8 Suponha que sua biblioteca tem uma função `Mrg` com parâmetros v, p, q, r que

funciona assim: ao receber um vetor v tal que $v[p\,..\,q]$ e $v[q{+}1\,..\,r]$ são crescentes, rearranja o vetor $v[p\,..\,r]$ em ordem crescente. Use `Mrg` para implementar o algoritmo Mergesort.

9.3 Desempenho do algoritmo

Quanto tempo a função `Mergesort` consome para ordenar um vetor $v[0\,..\,n{-}1]$? O número de elementos do vetor é reduzido aproximadamente à metade em cada invocação da função. Assim, o número total de "rodadas" é aproximadamente $\log_2 n$. Na primeira rodada, nosso problema original é reduzido a dois outros: ordenar

$$v[0\,..\,\tfrac{n}{2}{-}1] \quad \text{e} \quad v[\tfrac{n}{2}\,..\,n{-}1]$$

(para simplificar, estou supondo que n é uma potência de 2). Na segunda rodada temos quatro problemas: ordenar

$$v[0\,..\,\tfrac{n}{4}{-}1]\,, \quad v[\tfrac{n}{4}\,..\,\tfrac{n}{2}{-}1]\,, \quad v[\tfrac{n}{2}\,..\,\tfrac{3n}{4}{-}1] \quad \text{e} \quad v[\tfrac{3n}{4}\,..\,n{-}1]\,.$$

E assim por diante. O tempo total que `Intercala` gasta em cada rodada é proporcional a n (veja Exercício 9.3.1). Conclusão: `Mergesort` consome tempo proporcional a

$$n \log_2 n\,.$$

Isto é bem melhor que o tempo proporcional a n^2 gasto pelos algoritmos elementares do Capítulo 8. (Na prática, `Mergesort` só é realmente mais rápido que os algoritmos do Capítulo 8 quando n é suficientemente grande, uma vez que a constante de proporcionalidade na expressão "proporcional a" é maior no caso do `Mergesort`.)

Suponha que `Mergesort` consome t milissegundos para ordenar n números. Então a mesma função consumirá menos que $32t$ milissegundos para ordenar $16n$ números e menos que $352t$ milissegundos para ordenar $128n$ números. (Estamos supondo $n \geq 16$ em todas as estimativas.) Compare isso com o desempenho dos algoritmos elementares do Capítulo 8: se um daqueles algoritmos consumir t milissegundos para ordenar n números, consumirá $256t$ milissegundos para ordenar $16n$ números e $16384t$ milissegundos para ordenar $128n$ números.

Exercícios

9.3.1 Mostre que o consumo total de tempo de `Intercala` é proporcional a n em cada "rodada" de `Mergesort`.

9.3.2 Invocações repetidas de malloc. Durante uma execução de `Mergesort` (Seção 9.2), a função `Intercala` é invocada muitas vezes e cada execução de `Intercala` invoca as funções `malloc` e `free`. Para evitar as repetidas execuções de `malloc` e `free`, escreva uma versão da função `Mergesort` que incorpore o código da função de intercalação e invoque `malloc` e `free` uma só vez.

9.3.3 Overflow aritmético. Se o número de elementos do vetor estiver próximo de `INT_MAX` (veja Seção K.5), a execução da função `Mergesort` pode descarrilar, em virtude de um *overflow* aritmético, ao calcular o valor da expressão `(`p`+`r`)/2`. Como evitar isso? (Veja Exercício 7.5.3.)

9.3.4 Projeto de programação. Escreva um programa para comparar experimentalmente o desempenho da função `Mergesort` com o das funções `Inserção` e `Seleção` do Capítulo 8. (Para a fase de testes, escreva uma pequena função que verifique se sua implementação do Mergesort está produzindo uma ordenação correta do vetor.) Use um vetor aleatório (veja Apêndice I) para fazer os testes.

9.3.5 Ordem decrescente. Escreva uma versão do algoritmo Mergesort que rearranje um vetor $v[p\,..\,r{-}1]$ em ordem decrescente. (Será preciso reescrever o algoritmo da intercalação.)

9.3.6 Animações. Veja animações do algoritmo Mergesort nas páginas de Harrison [8] e Morin [13].

9.3.7 Leia o verbete *Merge sort* na Wikipedia [21].

9.4 Versão iterativa

Na versão iterativa do algoritmo Mergesort, cada iteração intercala dois "blocos" de b elementos: o primeiro bloco com o segundo, o terceiro com o quarto etc. A variável b assume os valores $1, 2, 4, 8, \ldots$

0				p		$p{+}b$		$p{+}2b$		$n{-}1$
111	999	222	999	333	888	444	777	555	666	555

Figura 9.3: Início da uma iteração da função `MergesortI` com $b = 2$.

```
/* Rearranja o vetor v[0..n-1] em ordem crescente. */
void MergesortI (int n, int v[]) {
   int p, r;
   int b = 1;
```

```
   while (b < n) {
      p = 0;
      while (p + b < n) {
         r = p + 2*b;
         if (r > n) r = n;
         Intercala (p, p+b, r, v);
         p = p + 2*b;
      }
      b = 2*b;
   }
}
```

Exercícios

9.4.1 Segmentos crescentes maximais. A versão iterativa do Mergesort começa por quebrar o vetor original em segmentos de comprimento 1. Quem sabe é melhor quebrar o vetor em seus segmentos crescentes maximais (os segmentos crescentes maximais de 1 2 3 0 2 4 6 4 5 6 7 8 9, por exemplo, são 1 2 3, 0 2 4 6 e 4 5 6 7 8 9). Escreva uma versão do Mergesort baseada nesta ideia.

9.4.2 Ordenação de strings. Escreva uma versão do algoritmo Mergesort que coloque um vetor de strings em ordem lexicográfica (veja Seção G.3).

9.4.3 Listas encadeadas. Escreva uma versão do algoritmo Mergesort que rearranje uma lista encadeada de modo que ela fique em ordem crescente. Sua função não deve alocar novas células na memória. (Veja Exercícios 9.1.10 e 4.7.7.) Faça duas versões: uma recursiva e uma iterativa.

Capítulo 10

Ordenação: algoritmo Heapsort

Heap: montão, amontoado, pilha.
— *Dicionário Michaelis*

O algoritmo Heapsort [22] resolve o problema da ordenação introduzido na Seção 8.1, ou seja, rearranja um vetor em ordem crescente. Para simplificar ligeiramente a descrição do algoritmo, suporemos neste capítulo que os índices do vetor são $1\,..\,n$ e não $0\,..\,n{-}1$.

O algoritmo Heapsort é bem mais rápido que os algoritmos elementares do Capítulo 8 e, ao contrário do Mergesort do Capítulo 9, não requer um vetor auxiliar.

10.1 Heap

O segredo do algoritmo Heapsort é uma estrutura de dados conhecida como *heap*.[1] Há dois "sabores" dessa estrutura: o max-heap e o min-heap. Apenas o primeiro será usado neste capítulo. Um **max-heap** é um vetor $v[1\,..\,m]$ tal que[2]

$$v[\lfloor \tfrac{1}{2} f \rfloor] \;\geq\; v[f]$$

para $f = 2, 3, \ldots, m$. (Num *min*-heap temos "$\leq$" no lugar de "$\geq$".) Segue imediatamente da definição que $v[1]$ é um elemento máximo do max-heap.

A estrutura de um heap fica mais clara se o conjunto de índices $1\,..\,m$ for entendido como um árvore binária (veja Capítulo 14):

[1] A palavra *heap* também designa a parte da memória do computador usada para alocação dinâmica, mas este significado não tem nenhuma relação com o conceito que estamos introduzindo aqui.

[2] Se f é par então $\lfloor f/2 \rfloor = f/2$, senão $\lfloor f/2 \rfloor = (f-1)/2$.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
999	888	666	333	777	555	555	333	222	111	444	111	222	444	111

Figura 10.1: Max-heap $v[1\,..\,15]$. Observe que $v[5] \geq v[10]$ e $v[5] \geq v[11]$.

a. o índice 1 é a **raiz** da árvore;
b. o **pai** de um índice f é $\lfloor \frac{1}{2} f \rfloor$;
c. o **filho esquerdo** de um índice p é $2p$ e o **filho direito** é $2p + 1$.

(É claro que 1 não tem pai, que o filho esquerdo de p só existe se $2p \leq m$ e que o filho direito de p só existe se $2p + 1 \leq m$.) Nesta árvore de índices, o *valor* do índice p é $v[p]$. Podemos dizer então que um vetor é um max-heap se todo pai é pelo menos tão valioso quanto qualquer dos seus dois filhos.

														1												
													2	3												
												4	5	6	7											
											8	9	10	11	12	13	14	15								
									16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31		
32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56		

Figura 10.2: A figura representa um vetor $v[1\,..\,56]$ em "camadas". O vetor é definido por $v[i] = i$ e portanto está longe de ser um max-heap. Cada filho está na camada imediatamente inferior à do pai. Cada camada, exceto talvez a última, tem duas vezes mais elementos que a camada anterior. Nesta representação, o número de camadas de $v[1\,..\,m]$ é exatamente $1 + \lfloor \log_2 m \rfloor$. (Veja Exercício 1.2.4.)

Exercícios

10.1.1 Mostre que todo vetor decrescente é um max-heap. Mostre que a recíproca não é verdadeira.

10.1.2 O vetor 161 41 101 141 71 91 31 21 81 17 16 é um max-heap?

10.1.3 Escreva uma função que decida se um vetor $v[1\,..\,m]$ é ou não um max-heap.

10.1.4 Mostre que $v[1\,..\,m]$ é um max-heap se e somente se valem as seguintes relações: (1) $v[p] \geq v[2p]$ para cada índice p tal que $2p \leq m$ e (2) $v[p] \geq v[2p+1]$ para cada índice p tal que $2p + 1 \leq m$.

10.1.5 Suponha que $v[1\,..\,m]$ é um max-heap. Mostre que $v[1] \geq v[j]$ para $j = 2, \ldots, m$.

10.1.6 Suponha que $v[1\,..\,2^k{-}1]$ é um max-heap. Mostre que mais da metade dos elementos do vetor está na última "camada" do max-heap, ou seja, em $v[2^{k-1}\,..\,2^k{-}1]$.

10.1.7 Suponha que $v[1\,..\,m]$ é um max-heap. Sejam i e j dois índices tais que $i < j$ e $v[i] < v[j]$. Se os valores de $v[i]$ e $v[j]$ forem trocados, $v[1\,..\,m]$ continuará sendo um max-heap? Repita o exercício sob a hipótese $v[i] > v[j]$.

10.2 Inserção em um heap

É fácil inserir um novo elemento em um max-heap de tal forma que a estrutura continue sendo um max-heap: basta "subir" em direção à raiz do heap à procura de um lugar apropriado para o novo elemento. A função abaixo insere $v[m{+}1]$ no max-heap $v[1\,..\,m]$:

```
/* Esta função recebe um max-heap v[1..m] e transforma
 * v[1..m+1] em max-heap. */
void InsereEmHeap (int m, int v[]) {
   int f = m+1;
   while /*X*/ (f > 1 && v[f/2] < v[f]) {
      int t = v[f/2]; v[f/2] = v[f]; v[f] = t;
      f = f/2;
   }
}
```

(É claro que o valor da expressão "`f/2`" é $\lfloor \frac{1}{2} f \rfloor$.) No início de cada iteração, ou seja, a cada passagem pelo ponto `X`, $v[1\,..\,m{+}1]$ é uma permutação do vetor original e a relação

$$v[\lfloor \tfrac{1}{2} i \rfloor] \geq v[i]$$

vale para todo i em $2\,..\,m{+}1$ que seja diferente de f. Em virtude desses invariantes, $v[1\,..\,m{+}1]$ é um max-heap no início da última iteração.

Desempenho. A função `InsereEmHeap` é muito rápida. Como o valor da variável f começa em $m + 1$ e é reduzido à metade em cada iteração, a função consome no máximo

$$\log_2(m + 1)$$

unidades de tempo. (O valor de cada unidade de tempo pode ser ligeiramente reduzido se reescrevermos o código como no Exercício 10.2.1.)

1	2	3	4	5	6	7	8	9	10	11	12	13	14
98	97	96	95	94	93	92	91	90	89	87	86	85	99
98	97	96	95	94	93	99	91	90	89	87	86	85	92
98	97	99	95	94	93	96	91	90	89	87	86	85	92
99	97	98	95	94	93	96	91	90	89	87	86	85	92

Figura 10.3: Inserção de $v[14]$ no max-heap $v[1\,..\,13]$. Cada linha da figura mostra o resultado de uma iteração.

Exercícios

10.2.1 Por que a seguinte implementação da função `InsereEmHeap` é ligeiramente mais eficiente que a dada no texto?

```
int p, f = m+1;
while (f > 1 && v[p = f/2] < v[f]) {
   int t = v[p]; v[p] = v[f]; v[f] = t;
   f = p; }
```

10.2.2 Escreva uma versão recursiva da função `InsereEmHeap`.

10.2.3 Construção de um max-heap. Escreva uma função eficiente que rearranje um vetor arbitrário de modo a transformá-lo em um max-heap. (Sugestão: use a função `InsereEmHeap`.)

10.2.4 Critique a seguinte ideia: para transformar um vetor arbitrário em max-heap, basta colocá-lo em ordem decrescente.

10.3 Um algoritmo auxiliar

Digamos que um vetor $v[1\,..\,m]$ é um **quase max-heap** se $v[\lfloor \frac{1}{2} f \rfloor] \geq v[f]$ para $f = 4, 5, \ldots, m$. Para transformar um quase max-heap em max-heap basta "sacudir" o vetor até que $v[1]$ "desça" para sua posição correta.

```
/* Rearranja um quase max-heap v[1..m] de modo a
 * transformá-lo em um max-heap. */
```

```
void SacodeHeap (int m, int v[]) {
   int t, f = 2;
   while /*X*/ (f <= m) {
      if (f < m && v[f] < v[f+1])  ++f;
      if (v[f/2] >= v[f])  break;
      t = v[f/2]; v[f/2] = v[f]; v[f] = t;
      f *= 2;
   }
}
```

(O primeiro `if` dentro do `while` faz com que f seja o filho mais valioso de $\lfloor f/2 \rfloor$.) Os invariantes do processo iterativo são simples: a cada passagem pelo ponto `X`, $v[1\,..\,m]$ é uma permutação do vetor original e

$$v[\lfloor \tfrac{1}{2} i \rfloor] \;\geq\; v[i]$$

para todo i em $2\,..\,m$ que seja diferente de f e de $f{+}1$.

A análise da última iteração é complicada pelo fato de que o processo iterativo pode ser interrompido em dois pontos diferentes. Suponha que estamos na última passagem pelo ponto `X`. Se tivermos $f > m$, a desigualdade $v[\lfloor \frac{1}{2} i \rfloor] \geq v[i]$ vale para todo i sem exceções e portanto $v[1\,..\,m]$ é um max-heap. Se $f = m$ então o processo iterativo terminará no `break` com $v[\lfloor \frac{f}{2} \rfloor] \geq v[f]$, donde o vetor $v[1\,..\,m]$ é um max-heap. Finalmente, se $f < m$ então o processo iterativo terminará no `break` com $v[\lfloor \frac{f}{2} \rfloor] \geq v[f]$ e $v[\lfloor \frac{f+1}{2} \rfloor] \geq v[f{+}1]$ e portanto $v[1\,..\,m]$ é um max-heap.

Desempenho. A função `SacodeHeap` é muito rápida. Como o valor de f pelo menos dobra a cada iteração, a função não consome mais que

$$\log_2 m$$

unidades de tempo. (O valor de cada unidade de tempo pode ser reduzido se reescrevermos o código como no Exercício 10.3.2.)

Exercícios

10.3.1 Mostre que os invariantes da função `SacodeHeap` valem no início da primeira iteração. Supondo que os invariantes valem no início de uma iteração, mostre que elas continuam válidas no início da iteração seguinte.

10.3.2 Verifique que a seguinte implementação da função `SacodeHeap` é ligeiramente mais eficiente que a dada no texto:

```
int p = 1, f = 2, t = v[1];
while (f <= m) {
   if (f < m && v[f] < v[f+1]) ++f;
   if (t >= v[f]) break;
   v[p] = v[f];
   p = f; f *= 2; }
v[p] = t;
```

10.3.3 Por que a seguinte versão da função `SacodeHeap` é incorreta?

```
p = 1, f = 2;
while (f <= m) {
   if (v[p] < v[f]) {
      t = v[p], v[p] = v[f], v[f] = t;
      p = f, f = 2*p; }
   else {
      if (f < m && v[p] < v[f+1]) {
         t = v[p], v[p] = v[f+1], v[f+1] = t;
         p = f+1, f = 2*p; }
      else break; } }
```

10.3.4 Escreva uma versão recursiva da função `SacodeHeap`.

10.4 O algoritmo Heapsort

O algoritmo Heapsort usa as funções `InsereEmHeap` e `SacodeHeap` para rearranjar um vetor $v[1..n]$ em ordem crescente. O algoritmo tem duas fases: a primeira transforma o vetor dado em um max-heap; a segunda usa a estrutura do max-heap para rearranjar o vetor em ordem crescente.

```
/* Rearranja o vetor v[1..n] de modo que ele fique
 * crescente. */
void Heapsort (int n, int v[]) {
   int m;
   for (m = 1; m < n; m++)
      InsereEmHeap (m, v);
   for (m = n; /*X*/ m > 1; m--) {
      int t = v[1]; v[1] = v[m]; v[m] = t;
      SacodeHeap (m-1, v);
   }
}
```

O invariante do primeiro processo iterativo é simples: no início de cada

iteração $v[1\,..\,m]$ é um max-heap. Agora considere o segundo processo iterativo. A cada passagem pelo ponto `X`, valem os seguintes invariantes:

1. $v[1\,..\,n]$ é uma permutação do vetor original,
2. o vetor $v[1\,..\,m]$ é um max-heap,
3. $v[i] \leq v[j]$ para todo i em $1\,..\,m$ e todo j em $m{+}1\,..\,n$ e
4. o vetor $v[m{+}1\,..\,n]$ está em ordem crescente.

No início da última iteração (quando m vale 1), em virtude dos invariantes 3 e 4, o vetor $v[1\,..\,n]$ é crescente.

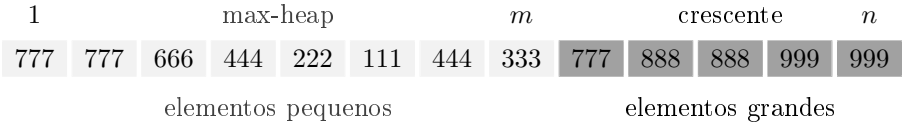


Figura 10.4: Início de uma iteração da segunda fase da função `Heapsort`.

Exercícios

10.4.1 Use a função `Heapsort` para ordenar o vetor 16 15 14 13 12 11 10 9 8 7 6 5 4 .

10.4.2 Suponha que o vetor $v[1\,..\,n]$ é um max-heap. O seguinte fragmento de código rearranja o vetor em ordem crescente?

```
for (m = n; m >= 2; m--) {
   int x = v[1];
   for (j = 1; j < m; ++j) v[j] = v[j+1];
   v[m] = x; }
```

10.4.3 Verifique os invariantes da função `Heapsort`.

10.5 Desempenho do algoritmo

Cada invocação de `InsereEmHeap` consome no máximo $\log_2(m+1)$ unidades de tempo. Logo, o consumo de tempo do primeiro processo iterativo da função `Heapsort` não passa de $\sum_{m=1}^{n-1} \log_2(m+1)$ unidades de tempo.

Agora considere o segundo processo iterativo. Como a função `SacodeHeap` consome no máximo $\log_2 m$ unidades de tempo, o processo todo consome no

máximo $\sum_{m=1}^{n} \log_2 m$ unidades de tempo. (Essas unidade de tempo não são necessariamente iguais às da primeira fase do algoritmo.) Resumindo: a função `Heapsort` não consome mais que

$$n \log_2 n$$

unidades de tempo.

Exercícios

10.5.1 Ordem decrescente. Escreva uma versão do algoritmo Heapsort que rearranje um vetor $v[1..n]$ de modo que ele fique em ordem decrescente.

10.5.2 Ordenação de strings. Escreva uma versão do algoritmo Heapsort que coloque um vetor de strings em ordem lexicográfica (veja Seção G.3).

10.5.3 Animações. Veja animações do algoritmo Heapsort nas páginas de Harrison [8] e Morin [13].

10.5.4 Leia o verbete *Heapsort* na Wikipedia [21].

Capítulo 11

Ordenação: algoritmo Quicksort

O algoritmo Quicksort [9] resolve o problema da ordenação introduzido na Seção 8.1, ou seja, rearranja um vetor $v[0\,..\,n{-}1]$ de modo que ele fique crescente. Em geral, o algoritmo é muito mais rápido que os algoritmos elementares do Capítulo 8, mas pode ser tão lento quanto aqueles para certas instâncias especiais do problema.

Usaremos duas abreviaturas: a expressão "$v[h\,..\,j] \leq x$" será usada como abreviatura de "$v[i] \leq x$ para todo i no conjunto de índices $h\,..\,j$" e a expressão "$v[h\,..\,j] \leq v[k\,..\,m]$" será interpretada como "$v[i] \leq v[l]$ para todo i no conjunto $h\,..\,j$ e todo l no conjunto $k\,..\,m$".

11.1 O problema da separação

O núcleo do algoritmo Quicksort é o seguinte problema da separação, que formularemos de maneira propositalmente vaga:

> rearranjar um vetor $v[p\,..\,r]$ de modo que os elementos pequenos fiquem todos do lado esquerdo e os grandes fiquem todos do lado direito.

Gostaríamos que os dois lados tivessem aproximadamente o mesmo número de elementos, mas estamos dispostos a aceitar resultados menos equilibrados. De todo modo, é importante que a separação não resulte degenerada, deixando um dos lados vazio. A dificuldade está em construir um algoritmo que resolva o problema de maneira rápida e não use um vetor auxiliar.

O problema da separação admite várias formulações concretas. Eis uma primeira: rearranjar $v[p\,..\,r]$ de modo que tenhamos

$$v[p\,..\,j] \;\leq\; v[j{+}1\,..\,r] \tag{11.1}$$

para algum j em $p\,..\,r{-}1$. Outra formulação: rearranjar $v[p\,..\,r]$ de modo que

$$v[p\,..\,j{-}1] \;\le\; v[j] \;<\; v[j{+}1\,..\,r] \tag{11.2}$$

para algum j em $p\,..\,r$. Este capítulo usa a segunda formulação; outras formulações serão mencionadas nos exercícios.

Exercícios

11.1.1 Escreva uma função que rearranje um vetor $v[p\,..\,r]$ de números inteiros de modo que os elementos negativos e nulos fiquem à esquerda e os positivos fiquem à direita. Em outras palavras, rearranje o vetor de modo que tenhamos $v[p\,..\,j{-}1] \le 0$ e $v[j\,..\,r] > 0$ para algum j em $p\,..\,r{+}1$. (Faz sentido exigir que j esteja em $p\,..\,r$?) Procure escrever uma função eficiente que não use vetor auxiliar. Repita o exercício depois de trocar "$v[j\,..\,r] > 0$" por "$v[j\,..\,r] \ge 0$".

11.1.2 Digamos que um vetor $v[p\,..\,r]$ está *arrumado* se existe j em $p\,..\,r$ que satisfaz (11.2). Escreva um algoritmo que decida se $v[p\,..\,r]$ está arrumado. Em caso afirmativo, o seu algoritmo deve devolver o valor de j.

11.1.3 Critique a solução do problema da separação dada abaixo. Qual das formulações concretas do problema ela satisfaz?

```
int w[1000], c, i, j, k;
c = v[p]; i = p; j = r;
for (k = p+1; k <= r; k++)
   if (v[k] <= c) w[i++] = v[k];
   else w[j--] = v[k];
w[j] = c;
for (k = p; k <= r; k++) v[k] = w[k];
return j;
```

11.1.4 Um programador inexperiente afirma que a seguinte função resolve a formulação (11.1) do problema da separação. Mostre uma instância em que a função não dá o resultado esperado.

```
int q, i, j, t;
i = p; q = (p + r)/2; j = r;
do {
   while (v[i] < v[q]) i++;
   while (v[j] > v[q]) j--;
   if (i <= j) {
      t = v[i], v[i] = v[j], v[j] = t;
      i++, j--; }
} while (i < j);
return i;
```

11.2 Algoritmo da separação

A seguinte função é uma solução eficiente da formulação (11.2) do problema da separação.[1] Ela começa por escolher um "pivô" c que definirá o significado de *pequeno* e *grande*: os elementos do vetor que forem maiores que c serão considerados grandes e os demais serão considerados pequenos.

```
/* Recebe um vetor v[p..r] com p <= r. Rearranja os
 * elementos do vetor e devolve j em p..r tal que
 * v[p..j-1] <= v[j] < v[j+1..r]. */
int Separa (int p, int r, int v[]) {
   int c, j, k, t;
   c = v[r]; j = p;
   for (k = p; /*A*/ k < r; k++)
      if (v[k] <= c) {
         t = v[j], v[j] = v[k], v[k] = t;
         j++;
      }
   v[r] = v[j], v[j] = c;
   return j;
}
```

No início de cada iteração, ou seja, a cada passagem pelo ponto `A`, valem os seguintes invariantes:

1. $v[p\,..\,r]$ é uma permutação do vetor original,
2. $v[p\,..\,j{-}1] \le c < v[j\,..\,k{-}1]$, $v[r] = c$ e
3. $p \le j \le k \le r$.

Na última passagem por `A` teremos $k = r$ e portanto $v[j\,..\,r{-}1] > c$. Assim, depois da troca de $v[j]$ com $v[r]$, teremos $v[p\,..\,j{-}1] \le v[j] < v[j{+}1\,..\,r]$.

Desempenho do algoritmo da separação. O consumo de tempo da função `Separa` é proporcional ao número de iterações. Como o número de iterações é $r - p + 1$, podemos dizer que o consumo de tempo é proporcional ao número de elementos do vetor.

[1] Esta versão da função consta do livro de Cormen *et al.* [5]. Compare-a com a função `RemoveZeros` na Seção 3.5.

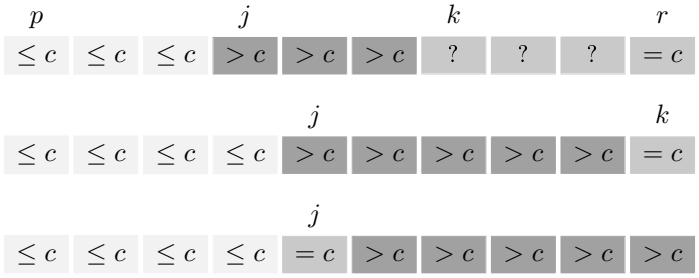


Figura 11.1: A parte superior da figura mostra a configuração no início de uma iteração qualquer da função `Separa`. A parte média mostra a configuração na última passagem pelo ponto `A`. A parte inferior mostra o resultado final de `Separa`.

Exercícios

11.2.1 A função `Separa` produz o resultado correto quando $p = r$?

11.2.2 Aplique a função `Separa` a um vetor cujos elementos são todos iguais. Aplique a função a um vetor cujos elementos só têm dois possíveis valores. Aplique a função a um vetor crescente e a um vetor decrescente.

11.2.3 Critique a seguinte variante da parte central do código da função `Separa`:

```
for (k = p; k < r; k++) {
   if (v[k] <= c) {
      if (j < k) t = v[j], v[j] = v[k], v[k] = t;
      j++; } }
if (j < r) v[r] = v[j], v[j] = c;
```

11.2.4 Critique a seguinte versão da função `Separa`. Quais os invariantes?

```
int c = v[p], i = p+1, j = r, t;
while (i <= j) {
   if (v[i] <= c) ++i;
   else {
      t = v[i], v[i] = v[j], v[j] = t;
      --j; } }
v[p] = v[j], v[j] = c;
return j;
```

11.2.5 A seguinte versão da função `Separa` aparece no livro de Gries [7]. Quais os invariantes do processo iterativo?

```
int c = v[p], i = p+1, j = r, t;
while (1) {
   while (i <= r && v[i] <= c) ++i;
   while (c < v[j]) --j;
   if (i >= j) break;
```

```
      t = v[i], v[i] = v[j], v[j] = t;
      ++i, --j; }
   v[p] = v[j], v[j] = c;
   return j;
```

11.2.6 Verifique que a seguinte versão da função `Separa` resolve a formulação (11.2) do problema da separação (veja Seção 11.1). Mostre que ela é equivalente à do Exercício 11.2.5. Quais são os invariantes no ponto `A`?

```
int c = v[p], i = p+1, j = r, t;
while (/*A*/ i <= j) {
   if (v[i] <= c) ++i;
   else if (c < v[j]) --j;
   else {
      t = v[i], v[i] = v[j], v[j] = t;
      ++i, --j; } } /* agora i == j+1 */
v[p] = v[j], v[j] = c;
return j;
```

11.2.7 Estabilidade. A função `Separa` produz um rearranjo estável (veja Seção 8.5) do vetor?

11.2.8 Escreva uma versão recursiva do algoritmo da separação.

11.2.9 Desafio. Escreva uma versão do algoritmo da separação que produza um índice j tal que $j - p$ fique entre $\frac{1}{4}(r-p)$ e $\frac{3}{4}(r-p)$.

11.3 Algoritmo Quicksort básico

Resolvido o problema da separação, podemos cuidar do Quicksort propriamente dito. O algoritmo usa a estratégia "dividir para conquistar" e poderia ser descrito vagamente como "um Mergesort ao contrário":

```
/* Esta função rearranja o vetor v[p..r], com p <= r+1,
 * de modo que ele fique em ordem crescente. */
void Quicksort (int p, int r, int v[]) {
   int j;
   if (p < r) {
      j = Separa (p, r, v);
      Quicksort (p, j - 1, v);
      Quicksort (j + 1, r, v);
   }
}
```

(Observe que a função está correta mesmo quando $p > r$, ou seja, quando o vetor está vazio.) Para ordenar um vetor $v[0..n-1]$, basta dizer `Quicksort (0, n - 1, v)`.

Exercícios

11.3.1 Que acontece se trocarmos "`if (p < r)`" por "`if (p != r)`" no código da função `Quicksort`?

11.3.2 No código da função `Quicksort`, que acontece se trocarmos a invocação "`Quicksort (p, j-1, v)`" por "`Quicksort (p, j, v)`"? Que acontece se trocarmos a invocação "`Quicksort (j+1, r, v)`" por "`Quicksort (j, r, v)`"?

11.3.3 Compare o código da função `Quicksort` com o da função `Mergesort` (Capítulo 9). Discuta as semelhanças e diferenças.

11.3.4 Submeta à função `Quicksort` o vetor 99 55 33 77 indexado por `1..4`. Teremos a seguinte sequência de invocações da função:

```
Quicksort (1, 4, v)
     Quicksort (1, 2, v)
          Quicksort (1, 0, v)
          Quicksort (2, 2, v)
     Quicksort (4, 4, v)
```

Repita o exercício com o vetor 55 44 22 11 66 33 indexado por `1..6`.

11.3.5 Reescreva a função `Quicksort` trocando a invocação de `Separa` pelo código da função.

11.3.6 Tail recursion. Mostre que a segunda invocação da função `Quicksort` pode ser eliminada se trocarmos o `if` por um `while` apropriado.

11.3.7 Escreva uma implementação do algoritmo Quicksort que evite aplicar a função a vetores com menos que dois elementos.

11.3.8 Separação reformulada. Suponha dada uma versão da função `Separa` que resolve a formulação (11.1) do problema da separação (veja Seção 11.1). Escreva uma variante do Quicksort que use essa versão de `Separa`.

11.3.9 Separação reformulada. Suponha dada uma versão da função `Separa` que rearranja o vetor $v[p..r]$ e devolve um índice i em $p..r$ tal que $v[p..i-1] \leq v[i] \leq v[i+1..r]$. Escreva uma variante do Quicksort que use essa versão de `Separa`.

11.3.10 Estabilidade. A função `Quicksort` produz uma ordenação estável (veja Seção 8.5)?

11.3.11 Versão iterativa. Escreva uma versão não recursiva do algoritmo Quicksort.

11.4 Desempenho do algoritmo

Quanto tempo a função `Quicksort` consome para ordenar um vetor $v[0\,..\,n{-}1]$? O consumo de tempo é proporcional ao número de comparações entre elementos do vetor. Se o índice j devolvido por `Separa` estiver sempre mais ou menos a meio caminho entre p e r, o número de comparações será aproximadamente

$$n \log_2 n\,.$$

Caso contrário, o número de comparações estará na ordem de n^2. (Isso acontece, por exemplo, se o vetor já estiver ordenado ou quase ordenado.) Portanto, o pior caso do Quicksort não é melhor que o dos algoritmos elementares do Capítulo 8.

Felizmente, o pior caso é raro. O consumo de tempo *médio* do `Quicksort` é proporcional a $n \log_2 n$. (Veja Cormen *et al.* [5]).

Exercícios

11.4.1 Aplique a função `Quicksort` a um vetor crescente com n elementos. Mostre que o número de comparações entre elementos do vetor é proporcional a n^2. (Veja Exercício 11.2.2.) Repita o exercício com vetor decrescente.

11.4.2 Ordenação de strings. Escreva uma versão do algoritmo Quicksort que coloque um vetor de strings em ordem lexicográfica (veja Seção G.3).

11.4.3 Ordenação de lista encadeada. Escreva uma versão do algoritmo Quicksort que rearranje uma lista encadeada de modo que ela fique em ordem crescente. Sua função não deve alocar novas células na memória.

11.5 Altura da pilha de execução do Quicksort

Na versão básica do Quicksort (veja Seção 11.3), o código cuida imediatamente do subvetor $v[p\,..\,j{-}1]$ e trata do subvetor $v[j{+}1\,..\,r]$ somente depois que $v[p\,..\,j{-}1]$ estiver ordenado. Dependendo do valor de j nas sucessivas invocações da função, a pilha de execução (veja Seção 6.5) pode crescer muito, atingindo altura igual ao número de elementos do vetor. (Isso acontece, por exemplo, se o vetor estiver em ordem decrescente.) O fenômeno não afeta o consumo de tempo do algoritmo, mas pode esgotar o espaço de memória. Para controlar o crescimento da pilha de execução é preciso tomar duas providências:

1. cuidar primeiro do *menor* dos subvetores $v[p\,..\,j{-}1]$ e $v[j{+}1\,..\,r]$ e
2. eliminar a segunda invocação recursiva da função (veja Exercício 11.3.6 acima).

Se adotarmos estas providências, o tamanho do subvetor que está no topo da pilha de execução será menor que a metade do tamanho do subvetor que está logo abaixo na pilha. De modo mais geral, o subvetor que está em qualquer das posições da pilha de execução será menor que metade do subvetor que está imediatamente abaixo. Assim, se a função for aplicada a um vetor com n elementos, a altura da pilha não passará de $\log_2 n$.

```
/* Esta função rearranja o vetor v[p..r], com p <= r+1,
 * de modo que ele fique em ordem crescente. */
void QuickSort (int p, int r, int v[]) {
   int j;
   while (p < r) {
      j = Separa (p, r, v);
      if (j - p < r - j) {
         QuickSort (p, j-1, v);
         p = j + 1;
      } else {
         QuickSort (j+1, r, v);
         r = j - 1;
      }
   }
}
```

Exercícios

11.5.1 A seguinte versão do algoritmo Quicksort consta da primeira edição do livro de Cormen *et al.* [5]. Verifique que ela se baseia na formulação (11.1) do problema da separação (Seção 11.1). Dê os invariantes do `while` externo.

```
void QuicksortCLR (int p, int r, int v[]) {
   int c = v[p], i = p-1, j = r+1, t;
   if (p < r) {
      while (1) {
         do --j; while (v[j] > c);
         do ++i; while (v[i] < c);
         if (i >= j) break;
         t = v[i], v[i] = v[j], v[j] = t; }
      QuicksortCLR (p, j, v);
      QuicksortCLR (j+1, r, v); } }
```

11.5.2 A versão abaixo do Quicksort é semelhante à do livro de Sedgewick [18]. Formule o problema da separação que esta versão resolve. Dê os invariantes do `while` externo.

```
void QuicksortS (int p, int r, int v[]) {
   int c = v[(p+r)/2], i = p, j = r, t;
   if (p < r) {
      while (i <= j) {
         while (v[i] < c) ++i;
         while (c < v[j]) --j;
         if (i <= j) {
            t = v[i], v[i] = v[j], v[j] = t;
            ++i, --j; } }
      QuicksortS (p, j, v);
      QuicksortS (i, r, v); } }
```

11.5.3 QUICKSORT ALEATORIZADO. Para tentar evitar o comportamento de pior caso da função `Separa`, podemos escolher o pivô aleatoriamente, recorrendo à função `InteiroAleatório` (veja Apêndice I):

```
int SeparaAleatorizado (int p, int r, int v[]) {
   int i, t;
   i = InteiroAleatório (p, r);
   t = v[p], v[p] = v[i], v[i] = t;
   return Separa (p, r, v); }
```

Use esta função para escrever uma implementação aleatorizada do algoritmo Quicksort.

11.5.4 ANIMAÇÕES. Veja animações do algoritmo Quicksort nas páginas de Harrison [8] e Morin [13].

11.5.5 Leia o verbete *Quicksort* na Wikipedia [21].

11.5.6 Familiarize-se com a função `qsort` da biblioteca `stdlib` (veja Seção K.1).

Capítulo 12

Algoritmos de enumeração

"Often it appears that there is no better way to solve a problem
than to try all possible solutions.
This approach, called exhaustive search, is almost always slow,
but sometimes it is better than nothing."

— I. Parberry, *Problems on Algorithms*

Para resolver certos problemas combinatórios, é necessário *enumerar* — ou seja, fazer uma lista de — todos os objetos de um determinado tipo. O número de objetos a enumerar é tipicamente muito grande, e portanto os algoritmos enumerativos consomem muito tempo. Algoritmos de enumeração estão relacionados com palavras-chave como *busca exaustiva*, *força bruta* e *backtracking*.

Este capítulo trata da enumeração de sequências de números naturais, mas as ideias também se aplicam à enumeração de outros tipos de objetos. Os algoritmos não são complexos, mas têm suas sutilezas. As versões recursivas são particularmente úteis e interessantes.

12.1 Enumeração de subsequências

Uma subsequência é o que sobra de uma sequência quando alguns de seus termos são apagados. Mais precisamente, uma **subsequência** de $s_1, s_2, \ldots, s_n$ é qualquer sequência da forma $s_{i_1}, s_{i_2}, \ldots, s_{i_k}$ onde $1 \leq i_1 < i_2 < \cdots < i_k \leq n$. Por exemplo, $2, 3, 5, 8$ é uma subsequência de $1, 2, 3, 4, 5, 6, 7, 8$.

> Problema: Enumerar todas as subsequências de $1, 2, \ldots, n$, ou seja, fazer uma lista em que cada subsequência aparece uma e uma só vez.

Há uma correspondência biunívoca óbvia entre as subsequências de $1, 2, \ldots, n$ e

```
1
1 2
1 2 3
1 2 3 4
1 2 4
1 3
1 3 4
1 4
2
2 3
2 3 4
2 4
3
3 4
4
```

Figura 12.1: Todas as subsequências não vazias de $1, 2, 3, 4$, em ordem lexicográfica.

os subconjuntos do conjunto $\{1, 2, \ldots, n\}$. Portanto, o número de subsequências de $1, 2, \ldots, n$ é 2^n. Este número aumenta explosivamente com n: ele dobra toda vez que n aumenta de uma unidade.

Nossas sequências serão armazenadas em vetores. A sequência $s_1, \ldots, s_k$, por exemplo, será armazenada num vetor $s[1\,..\,k]$. Com isso, as expressões "$s[i]$" e "s_i" serão consideradas sinônimas. A primeira é mais apropriada no código C, enquanto a segunda é mais apropriada no texto em português.

12.2 Subsequências em ordem lexicográfica

A ordem em que as subsequências de $1, 2, \ldots, n$ são enumeradas não é muito importante, mas certas ordens são mais naturais que outras. Uma das ordens mais naturais é a *lexicográfica* (esta é a ordem em que palavras aparecem em um dicionário, conforme Seção G.3). Uma sequência $r_1, r_2, \ldots, r_j$ é **lexicograficamente menor que** outra $s_1, s_2, \ldots, s_k$ se

1. $j < k$ e $r_1, \ldots, r_j$ é igual a $s_1, \ldots, s_j$ ou
2. existe i tal que $r_1, \ldots, r_{i-1}$ é igual a $s_1, \ldots, s_{i-1}$ e $r_i < s_i$.

A seguinte função gera uma lista de sequências em ordem lexicográfica:

```
/* Esta função recebe n >= 1 e imprime todas as
 * subsequências não vazias de 1,2,...,n
 * em ordem lexicográfica. */
```

```
void SubseqLex (int n) {
   int *s, k;
   s = malloc ((n+1) * sizeof (int));
   s[0] = 0; k = 0;
   while (1) {
      if (s[k] < n) {
         s[k+1] = s[k] + 1;
         k += 1;
      } else {
         s[k-1] += 1;
         k -= 1;
      }
      if (k == 0) break;
      imprima (s, k);
   }
   free (s);
}
```

Cada iteração começa com uma subsequência $s_1, s_2, \ldots, s_k$ de $1, 2, \ldots, n$ armazenada no vetor `s[1..k]`. A primeira iteração começa com a subsequência vazia. Cada iteração gera a sucessora de $s_1, s_2, \ldots, s_k$ na ordem lexicográfica. Se $s_1, s_2, \ldots, s_k$ não tiver sucessora, o processo termina.

Três detalhes da função `SubseqLex` merecem comentário. (1) A expressão `imprima(s, k)` não faz mais que imprimir o vetor `s[1..k]`. (2) A sentinela `s[0]` foi definida para que o comando `s[k-1] += 1` possa ser executado quando k vale 1, coisa que acontece somente na última iteração. (3) O vetor `s[1..k]` comporta-se como uma pilha (veja Capítulo 6), sendo k o índice do topo da pilha.

Figura 12.2: Vetor s no início de uma iteração de `SubseqLex` com argumento $n = 7$.

Exercícios

12.2.1 Analise a seguinte variante da função `SubseqLex`:

```
s[0] = 0; s[1] = 1; k = 1;
while (k >= 1) {
   imprima (s, k);
   if (s[k] < n) {
      s[k+1] = s[k] + 1; k += 1; }
   else {
      s[k-1] += 1; k -= 1; } }
```

12.2.2 Analise a seguinte versão alternativa da função `SubseqLex`:

```
s[1] = 1; k = 1;
imprima (s, 1);
while (s[1] < n) {
   if (s[k] < n) {
      s[k+1] = s[k] + 1; k += 1; }
   else {
      s[k-1] += 1; k -= 1; }
   imprima (s, k); }
```

12.2.3 Escreva uma função que imprima uma lista de todos os subconjuntos do conjunto $\{1, 2, \ldots, n\}$.

12.3 Versão recursiva do algoritmo

A versão recursiva de `SubseqLex` é muito interessante. A interface com o usuário fica a cargo da seguinte função-embalagem:

```
/* Recebe n >= 1 e imprime, em ordem lexicográfica,
 * todas as subsequências não vazias de 1,2,...,n. */
void SubseqLex2 (int n) {
   int *s;
   s = malloc ((n+1) * sizeof (int));
   SseqR (s, 0, 1, n);
   free (s);
}
```

O serviço pesado é todo executado pela função recursiva `SseqR`. Para cada valor de m, ela imprime todas as subsequências que incluem m e depois todas as que não incluem m.

```
void SseqR (int s[], int k, int m, int n) {
   if (m <= n) {
      s[k+1] = m;
      imprima (s, k+1);
      SseqR (s, k+1, m+1, n);  /* inclui m */
      SseqR (s, k,   m+1, n);  /* não inclui m */
   }
}
```

A explicação dada acima sobre o que faz (veja introdução do Capítulo 1) a função `SseqR` é muito vaga, pois omite o papel dos parâmetros s e k. Eis uma documentação precisa:

```
/* A função SseqR recebe s[1..k] e m e imprime,
 * em ordem lexicográfica, cada uma das subsequências
 * não vazias de m,...,n precedida do prefixo s[1..k].
 * Em outras palavras, imprime todas as sequências que
 * têm a forma s[1],..,s[k],t[k+1],..., sendo t[k+1],...
 * uma subsequência não vazia de m,...,n. */
```

Portanto, o cálculo da expressão `SseqR(s, 0, 1, n)` faz exatamente o que queremos: imprime todas as subsequências $t_1, t_2, \ldots$ de $1, 2, \ldots, n$ que têm pelo menos um termo.

```
2 4 7
2 4 7 8
2 4 7 8 9
2 4 7 9
2 4 8
2 4 8 9
2 4 9
```

Figura 12.3: Resultado de `SseqR(s,2,7,9)` com `s[1]` $= 2$ e `s[2]` $= 4$. A primeira linha é gerada por `imprima(s,3)`. As três linhas seguintes são geradas por `SseqR(s,3,8,9)`. As demais, por `SseqR(s,2,8,9)`.

Exercícios

12.3.1 Escreva uma função que receba um vetor estritamente crescente $s[1..k]$ representando uma subsequência $s_1, s_2, \ldots, s_k$ de $1, 2, \ldots, n$ e grave, no espaço alocado a s,

a próxima subsequência na ordem lexicográfica. A função deve devolver o número de termos ($k+1$ ou $k-1$) da nova subsequência.

12.4 Subsequências em ordem lexicográfica especial

A ordem lexicográfica "especial" dá preferência às subsequências mais longas. Eis a definição formal: uma sequência $r_1, r_2, \ldots, r_j$ **precede** outra $s_1, s_2, \ldots, s_k$ **na ordem lexicográfica especial** se

1. $j > k$ e $r_1, \ldots, r_k$ é igual a $s_1, \ldots, s_k$ ou
2. existe i tal que $r_1, \ldots, r_{i-1}$ é igual a $s_1, \ldots, s_{i-1}$ e $r_i < s_i$.

A seguinte função gera uma lista de sequências em ordem lexicográfica especial:

```
/* Recebe n >= 1 e imprime, em ordem lexicográfica especial,
 * todas as subsequências não vazias de 1,2,...,n. */
void SubseqLexEsp (int n) {
   int *s, k;
   s = malloc ((n+1) * sizeof (int));
   s[1] = 0; k = 1;
   while (1) {
      if (s[k] == n) {
         k -= 1;
         if (k == 0) break;
      } else {
         s[k] += 1;
         while (s[k] < n) {
            s[k+1] = s[k] + 1;
            k += 1;
         }
      }
      imprima (s, k);
   }
   free (s);
}
```

Na versão recursiva da função, convém imprimir também a subsequência vazia:

```
1  2  3  4
1  2  3
1  2  4
1  2
1  3  4
1  3
1  4
1
2  3  4
2  3
2  4
2
3  4
3
4
```

Figura 12.4: Todas as subsequências não vazias de $1, 2, 3, 4$, em ordem lexicográfica especial.

```
/* Recebe n >= 1 e imprime, em ordem lexicográfica
 * especial, todas as subsequências de 1,2,...,n. */
void SubseqLexEsp2 (int n) {
   int *s;
   s = malloc ((n+1) * sizeof (int));
   SseqEspR (s, 0, 1, n);
   free (s);
}
/* Esta função auxiliar recebe um vetor s[1..k] e imprime,
 * em ordem lexicográfica especial, todas as sequências
 * da forma s[1],...,s[k],t[k+1],... tais que
 * t[k+1],... é uma subsequência de m,m+1,...,n.
 * Em seguida, imprime a sequência s[1],...,s[k]. */
void SseqEspR (int s[], int k, int m, int n) {
   if (m > n) imprima (s, k);
   else {
      s[k+1] = m;
      SseqEspR (s, k+1, m+1, n);  /* inclui m */
      SseqEspR (s, k,   m+1, n);  /* não inclui m */
   }
}
```

De acordo com a documentação da função, o comando `SseqEspR(`s`, 0, 1, `n`)` imprime todas subsequências de $1, 2, \ldots, n$, como desejado.

```
2  4  7  8  9
2  4  7  8
2  4  7  9
2  4  7
2  4  8  9
2  4  8
2  4  9
2  4  9
2  4
```

Figura 12.5: Lista impressa por `SseqEspR(`s`,2,7,9)` supondo que s`[1]` $= 2$ e s`[2]` $= 4$.

Exercícios

12.4.1 Ordem militar. A lista abaixo (leia a coluna esquerda, depois a do meio, depois a direita) exibe as subsequências de $1, 2, 3, 4$ em "ordem militar". Analise esta ordem. Escreva uma função que imprima todas as subsequências de $1, 2, \ldots, n$ em ordem militar. Escreva duas versões: uma iterativa e uma recursiva.

```
1          1  3       1  2  3
2          1  4       1  2  4
3          2  3       1  3  4
4          2  4       2  3  4
1  2       3  4       1  2  3  4
```

12.4.2 Subset sum. Suponha que você emitiu cheques com valores $p_1, \ldots, p_n$ ao longo de um mês. No fim do mês, o banco informa que uma quantia T foi descontada de sua conta. Quais dos cheques foram descontados? Por exemplo, se $p = 61, 62, 63, 64$ e $T = 125$ então só há duas possibilidades: ou foram descontados os cheques 1 e 4 ou foram descontados os cheques 2 e 3. Esta é uma instância do problema *subset sum* (soma de subconjunto): dado um número T e um vetor $p[1\,..\,n]$, encontrar todas as subsequências $s_1, s_2, \ldots, s_k$ de $1, 2, \ldots, n$ para as quais $p[s_1] + \cdots + p[s_k] = T$. Escreva uma função que resolva o problema.

12.4.3 Combinações. Escreva uma função que imprima todas as subsequências de $1, 2, \ldots, n$ que têm exatamente k termos. (Isso corresponde aos subconjuntos de $\{1, 2, \ldots, n\}$ que têm exatamente k elementos.)

12.4.4 Permutações. Uma *permutação* da sequência $1, 2, \ldots, n$ é qualquer rearranjo desta sequência. Por exemplo, as seis permutações de $(1, 2, 3)$ são $(1, 2, 3)$, $(1, 3, 2)$, $(2, 1, 3)$, $(2, 3, 1)$, $(3, 1, 2)$ e $(3, 2, 1)$. Escreva uma função que imprima, exatamente uma vez, cada uma das $n!$ permutações de $1, 2, \ldots, n$.

12.4.5 Desarranjos. Um *desarranjo* da sequência $1, 2, \ldots, n$ é qualquer permutação desta sequência que muda *todos* os termos de posição. Em outras palavras, um desarranjo de $1, 2, \ldots, n$ é qualquer permutação $p_1, p_2, \ldots, p_n$ de $1, 2, \ldots, n$ tal que $p_i \neq i$ para todo i. Por exemplo, os nove desarranjos de $(1, 2, 3, 4)$ são $(2, 1, 4, 3)$, $(2, 3, 4, 1)$, $(2, 4, 1, 3)$, $(3, 1, 4, 2)$, $(3, 4, 1, 2)$, $(3, 4, 2, 1)$, $(4, 1, 2, 3)$, $(4, 3, 1, 2)$ e $(4, 3, 2, 1)$. Escreva uma função que imprima, exatamente uma vez, cada desarranjo de $1, 2, \ldots, n$.

12.4.6 PARTIÇÕES. Escreva uma função que imprima uma lista de todas as partições[1] do conjunto $\{1, 2, \ldots, n\}$ em m blocos não vazios. Uma tal partição pode ser representada por um vetor $p[1\,..\,n]$ com valores no conjunto $\{1, 2, \ldots, m\}$ dotado da seguinte propriedade: para cada i entre 1 e m, existe pelo menos um j tal que $p[j] = i$.

12.4.7 PROBLEMA DAS RAINHAS. É possível colocar 8 rainhas do jogo de xadrez sobre o tabuleiro de modo que nenhuma das rainhas possa atacar outra?

12.4.8 PASSEIO DO CAVALO. Suponha dado um tabuleiro de xadrez n-por-n. Determine se é possível que um cavalo do jogo de xadrez parta da posição $(1, 1)$ do tabuleiro e complete um passeio por todas as n^2 posições do tabuleiro em n^2-1 passos válidos. Por exemplo para um tabuleiro 5-por-5 uma solução do problema é indicada pela matriz abaixo.

1	6	15	10	21
14	9	20	5	16
19	2	7	22	11
8	13	24	17	4
25	18	3	12	23

Sugestão: Numere as casas do tabuleiro e examine todas as permutações de $1, 2, \ldots, n^2$. Para cada permutação, verifique se ela representa um passeio do cavalo.

12.4.9 Familiarize-se com a página *Combinatorial Object Server* de Ruskey [17].

[1] Uma *partição* de um conjunto X é qualquer coleção $\mathcal{P}$ de subconjuntos não vazios de X dotada da seguinte propriedade: todo elemento de X pertence a um e apenas um dos elementos de $\mathcal{P}$. Por exemplo, $\{\{1, 3\}, \{2, 4, 7\}, \{5\}, \{6, 8\}\}$ é uma partição de $\{1, 2, 3, 4, 5, 6, 7, 8\}$. O conjunto $\{6, 8\}$ é um dos *blocos* da partição.

Capítulo 13

Busca de palavras em um texto

Considere o problema de encontrar as ocorrências de uma dada sequência curta em outra longa. O problema, conhecido como *string searching* ou *string matching*, aparece naturalmente em um sem-número de aplicações, de editores de texto a genética computacional. O problema não é tão simples quanto parece se insistirmos em algoritmos eficientes.

13.1 O problema da busca

Um vetor $a[1..m]$ é **sufixo** de um vetor $b[1..k]$ se $m \leq k$ e $a[1..m] = b[k-m+1..k]$, ou seja, $a[1] = b[k-m+1]$, $a[2] = b[k-m+2]$, ..., $a[m] = b[k]$. Dizemos que um vetor $a[1..m]$ **ocorre em** um vetor $b[1..n]$ se existe k no intervalo $m..n$ tal que $a[1..m]$ é sufixo de $b[1..k]$.

Este capítulo estuda o problema de localizar todas as ocorrências de um vetor $a[1..m]$ em um vetor $b[1..n]$. Para simplificar, trataremos apenas de *contar* o número de ocorrências.

Problema: Encontrar o número de ocorrências de $a[1..m]$ em $b[1..n]$.

Suporemos que a e b são vetores de caracteres, embora o problema também faça sentido para outros tipos de dados. Diremos que o vetor a é uma **palavra** e que b é um **texto**.

```
typedef unsigned char *palavra;
typedef unsigned char *texto;
```

Com isso, nosso problema pode ser resumido assim: encontrar o número de ocorrências de uma dada palavra em um dado texto.

Para garantir que o número de ocorrências de a em b seja finito, suporemos

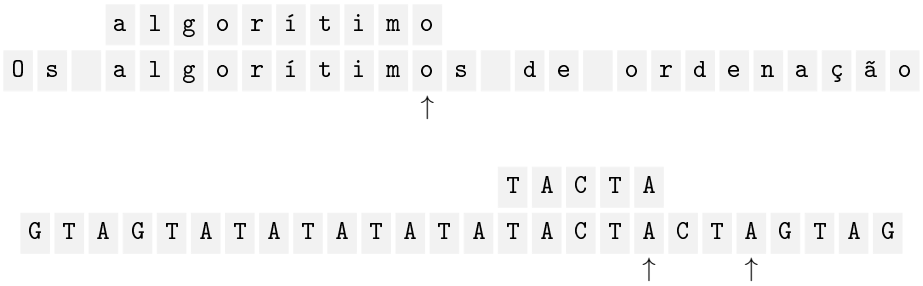


Figura 13.1: Dois exemplos de busca de uma palavra $a[1\,..\,m]$ em um texto $b[1\,..\,n]$. O primeiro sugere a busca de um erro de grafia em um documento. O segundo sugere a busca de um gene em um cromossomo. O sinal $\uparrow$ indica um índice k tal que $a[1\,..\,m]$ é sufixo de $b[1\,..\,k]$. No segundo exemplo há duas ocorrências sobrepostas de a em b.

que $m \geq 1$. Quanto a n, não há razão para excluir o caso $n = 0$. É claro que se $m > n$, o número de ocorrências de a em b é nulo.

Há duas questões simples que convém esclarecer desde já. Para procurar uma ocorrência de a em b, podemos varrer b da esquerda para a direita ou da direita para a esquerda. As duas alternativas são equivalentes, mas vamos adotar sempre a primeira: comparar a com $b[1\,..\,m]$, depois com $b[2\,..\,m{+}1]$, etc. Agora considere a questão de decidir se a é sufixo de $b[1\,..\,k]$ para um k fixo. A comparação de $a[1\,..\,m]$ com $b[k{-}m{+}1\,..\,k]$ pode ser feita da esquerda para a direita ou da direita para a esquerda. Em geral, as duas alternativas são equivalentes; mas um dos algoritmos que veremos adiante exige que a comparação seja feita na direção contrária à da varredura do texto. Por isso, adotaremos sempre comparação da direita para a esquerda: primeiro $a[m]$ com $b[k]$, depois $a[m{-}1]$ com $b[k{-}1]$, etc.

13.2 Algoritmo trivial

A seguinte função resolve nosso problema da maneira mais óbvia: pacientemente, procura a como sufixo de $b[1\,..\,m]$, depois como sufixo de $b[1\,..\,m{+}1]$, e assim por diante.

```
/* Recebe uma palavra a[1..m] e um texto b[1..n],
 * com m >= 1 e n >= 0, e devolve o número de
 * ocorrências de a em b. */
```

```
int trivial (palavra a, int m, texto b, int n) {
   int k, r, ocorrs;
   ocorrs = 0;
   for (k = m; k <= n; k++) {
      r = 0;
      while (r < m && a[m-r] == b[k-r]) r += 1;
      if (r >= m) ocorrs += 1;
   }
   return ocorrs;
}
```

A função `trivial` faz no máximo mn comparações entre os elementos dos dois vetores. Portanto, consome tempo proporcional a mn no pior caso. Gostaríamos de encontrar um algoritmo que não consumisse mais que $m+n$ unidades de tempo.

Exercício

13.2.1 Dê um exemplo em que o algoritmo trivial faz o maior número possível de comparações entre elementos de a e b. Descreva o exemplo com precisão.

13.3 Primeiro algoritmo de Boyer–Moore

Suponha que o conjunto a que pertencem os elementos de a e b é conhecido de antemão. Diremos que este conjunto é o **alfabeto** do problema. (O alfabeto pode ser, por exemplo, o conjunto de todos os 256 caracteres.) Neste caso, é possível construir um algoritmo melhor que o trivial. Para cada caractere c do alfabeto, defina o número `T1[`c`]` da seguinte maneira: se c está em a então

$$\texttt{T1[}c\texttt{]} \quad \text{é o menor } t \text{ em } 0\,..\,m{-}1 \text{ tal que } a[m-t] = c \tag{13.1}$$

e `T1[`c`]` $= m$ se c não está em a. Portanto, `T1[`c`]` corresponde à última ocorrência, no sentido esquerda–para–direita, do caractere c em a. Diremos que `T1[`c`]` é o *deslocamento* correspondente a c.

Suponha agora que c é igual a $b[k{+}1]$ para um determinado $k \geq m$. Se `T1[`c`]` $= 4$, por exemplo, então os caracteres $a[m]$, $a[m{-}1]$, $a[m{-}2]$, $a[m{-}3]$ são todos diferentes de c e portanto $a[1\,..\,m]$ não é sufixo de

$$b[1\,..\,k{+}1], \text{ nem de } b[1\,..\,k{+}2], \text{ nem de } b[1\,..\,k{+}3], \text{ nem de } b[1\,..\,k{+}4]$$

(supondo $k{+}4 \leq n$). De modo mais geral, se k é um índice em $m\,..\,n{-}1$ e c é o caractere $b[k{+}1]$, então os caracteres $a[m], a[m{-}1], a[m{-}2], \ldots, a[m-$`T1[`$c$`]`$+1]$

são todos diferentes de c e portanto a não é sufixo de

$$b[1\,..\,k{+}1],\ \text{nem de}\ b[1\,..\,k{+}2],\ \ldots\ ,\ \text{nem de}\ b[1\,..\,L]\,, \tag{13.2}$$

sendo L o menor dentre n e $k + \texttt{T1}[c]$.

Suponha que já determinamos o número, digamos N, de ocorrências de a em $b[1\,..\,k]$. Seja c o caractere $b[k{+}1]$. Se $k{+}\texttt{T1}[c] \geq n$ então, em virtude de (13.2), nosso problema está resolvido: o número de ocorrências de a em $b[1\,..\,n]$ é N. Se $k + \texttt{T1}[c] < n$ então podemos concluir, sem fazer quaisquer comparações adicionais, que o número de ocorrências de a em $b[1\,..\,k{+}\texttt{T1}[c]]$ é também N.

A ideia que acabamos de descrever é a base do primeiro algoritmo de Boyer–Moore [3]. A primeira fase do algoritmo, conhecida como *pré-processamento da palavra*, calcula a tabela `T1`. Na segunda fase, o algoritmo usa `T1` para procurar as ocorrências de a em b.

```
/* Recebe uma palavra a[1..m] e um texto b[1..n], com
 * m >= 1 e n >= 0, e devolve o número de ocorrências
 * de a em b. Supõe que cada elemento de a e b pertence
 * ao conjunto de caracteres 0..255. */
int BoyerMoore1 (palavra a, int m, texto b, int n) {
  int T1[256], i, k, r, ocorrs;
  /* pré-processamento da palavra a */
  for (i = 0; i < 256; i++) T1[i] = m;
  for (i = 1; i <= m; i++) T1[a[i]] = m - i;
  /* busca da palavra a no texto b */
  ocorrs = 0; k = m;
  while (k <= n) {
     r = 0;
     while (m-r >= 1 && a[m-r] == b[k-r]) r += 1;
     if (m-r < 1) ocorrs += 1;
     if (k == n) k += 1;
     else k += T1[b[k+1]] + 1;
  }
  return ocorrs;
}
```

O invariante principal é simples: no começo de cada iteração da fase de busca, o valor de `ocorrs` é o número de ocorrências de a em $b[1\,..\,k{-}1]$.

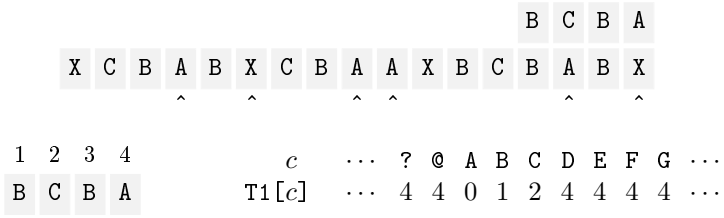


Figura 13.2: Primeiro algoritmo de Boyer–Moore aplicado à palavra `BCBA` e a um texto b. Assinalamos com ^ as únicas posições k em que a palavra é efetivamente comparada com $b[k{-}3\,..\,k]$. A parte inferior da figura exibe a tabela de deslocamentos `T1`.

Desempenho. O pré-processamento da palavra a consome m unidades de tempo. Na fase da busca, o consumo de tempo é proporcional ao número de comparações entre elementos de a e b. Tal como no algoritmo trivial, o número de tais comparações não passa de

$$m\,n$$

no pior caso. Mas o pior caso deste algoritmo é mais raro que o do algoritmo trivial. Assim, em geral, o número de comparações é bem menor que mn.

Exercícios

13.3.1 Mostre que a fase de pré-processamento na função `BoyerMoore1` preenche corretamente a tabela `T1`.

13.3.2 Dê um exemplo em que a função `BoyerMoore1` faz o maior número possível de comparações entre elementos de a e b. Descreva o exemplo com precisão.

13.3.3 Mostre que é possível eliminar o incômodo "`if (`k `==` n`)` k `+= 1; else`" no código da função `BoyerMoore1` com o auxílio de uma sentinela postada em b`[`n`+1]`.

13.3.4 Mostre que a seguinte variante da função `BoyerMoore1` está correta:

```
int T1[256], i, k, r, ocorrs;
for (i = 0; i < 256; i++) T1[i] = m;
for (i = 1; i < m; i++) T1[a[i]] = m - i;
ocorrs = 0; k = m;
while (k <= n) {
   r = 0;
   while (m - r >= 1 && a[m-r] == b[k-r]) r += 1;
   if (m - r < 1) ocorrs += 1;
   k += T1[b[k]]; }
return ocorrs;
```

13.4 Segundo algoritmo de Boyer–Moore

Para introduzir a ideia do segundo algoritmo de Boyer–Moore, considere o seguinte exemplo. Suponha que $a[i\,..\,m]$ é sufixo de $b[1\,..\,k]$. Suponha também que $i \geq 4$ e $a[i\,..\,m]$ não é sufixo de

$$a[1\,..\,m{-}1], \text{ nem de } a[1\,..\,m{-}2], \text{ nem de } a[1\,..\,m{-}3]\,.$$

Nestas condições, é fácil deduzir que a não é sufixo de

$$b[1\,..\,k{+}1], \text{ nem de } b[1\,..\,k{+}2], \text{ nem de } b[1\,..\,k{+}3]\,.$$

Para generalizar esta ideia, é necessário introduzir uma tabela `T2[1..m]`. Dado i no intervalo $1\,..\,m$, digamos que um índice j em $1\,..\,m{-}1$ é **bom para** i se

$$a[m{-}L\,..\,m] \;=\; a[j{-}L\,..\,j]\,,$$

sendo L o menor dentre $m-i$ e $j-1$. (Se $L = m-i$ então temos $a[i\,..\,m] = a[j{-}m{+}i\,..\,j]$; caso contrário, $m-j+1 \geq i$ e $a[m{-}j{+}1\,..\,m] = a[1\,..\,j]$.) Em outras palavras, j é bom para i se

$$a[i\,..\,m] \text{ é sufixo de } a[1\,..\,j] \quad \text{ ou } \quad a[1\,..\,j] \text{ é sufixo de } a[i\,..\,m]\,.$$

Para cada i no intervalo $1\,..\,m$, defina `T2[i]` da seguinte maneira: se existe um índice bom para i então

$$\texttt{T2[}i\texttt{]} \;\text{ é o menor } t \text{ em } 1\,..\,m{-}1 \text{ tal que } m-t \text{ é bom para } i\,; \qquad (13.3)$$

senão, $\texttt{T2[}i\texttt{]} = m$. (No exemplo acima, temos $\texttt{T2[}i\texttt{]} \geq 4$.) Diremos que `T2[i]` é o *deslocamento* correspondente a i. Com essa definição, se $a[i\,..\,m]$ é sufixo de $b[1\,..\,k]$ então podemos garantir que a não é sufixo de

	1	2	3	4	5	6
	C	A	A	B	A	A

i	6	5	4	3	2	1
T2[i]	1	3	6	6	6	6

	1	2	3	4	5	6	7	8
	B	A	-	B	A	.	B	A

i	8	7	6	5	4	3	2	1
T2[i]	3	3	6	6	6	6	6	6

	1	2	3	4	5	6	7	8	9	10	11
	B	A	-	B	A	*	B	A	*	B	A

i	11	10	9	8	7	6	5	4	3	2	1
T2[i]	3	3	3	3	3	9	9	9	9	9	9

Figura 13.3: Três exemplos de palavra $a[1\,..\,m]$ e correspondente tabela de deslocamentos `T2`. A tabela é usada pelo segundo algoritmo de Boyer–Moore.

$b[1\,..\,k{+}1]$, nem de $b[1\,..\,k{+}2]$, ... , nem de $b[1\,..\,k{+}\texttt{T2[}i\texttt{]}{-}1]$.

Portanto, nossa próxima tentativa deve decidir se a é sufixo de $b[1\,..\,k{+}\texttt{T2[}i\texttt{]}]$.

O segundo algoritmo de Boyer–Moore começa por pré-processar a palavra a para construir a tabela de deslocamentos `T2`. Em seguida, procura as ocorrências de a em b:

```
/* Recebe uma palavra a[1..m] com 1 <= m <= MAX e um texto
 * b[1..n] e devolve o número de ocorrências de a em b. */
int BoyerMoore2 (palavra a, int m, texto b, int n) {
  int T2[MAX], i, j, k, r, ocorrs;
  /* pré-processamento da palavra a */
  for (i = m; i >= 1; i--) {
     j = m-1; r = 0;
     while (m-r >= i && j-r >= 1)
        if (a[m-r] == a[j-r]) r += 1;
        else j -= 1, r = 0;
     T2[i] = m - j;
  }
  /* busca da palavra a no texto b */
  ocorrs = 0; k = m;
  while (k <= n) {
     r = 0;
     while (m-r >= 1 && a[m-r] == b[k-r]) r += 1;
     if (m-r < 1) ocorrs += 1;
     if (r == 0) k += 1;
     else k += T2[m-r+1];
  }
  return ocorrs;
}
```

No começo de cada iteração da fase de busca, o valor de `ocorrs` é o número de ocorrências de a em $b[1\,..\,k{-}1]$.

Desempenho. O pré-processamento da palavra consome m^2 unidades de tempo no pior caso. A fase de busca consome mn unidades de tempo no pior caso. No caso médio, entretanto, a fase de busca consome apenas n unidades de tempo.

Exercícios

13.4.1 Calcule a tabela `T2` no caso em que $a[1] = a[2] = \cdots = a[m]$. Calcule a tabela `T2` no caso em que os elementos de $a[1\,..\,m]$ são distintos dois a dois.

13.4.2 Mostre que a fase de pré-processamento na função `BoyerMoore2` preenche corretamente a tabela `T2`.

13.4.3 Dê um exemplo em que a função `BoyerMoore2` faz o maior número possível de comparações entre elementos de a e b. Descreva o exemplo com precisão.

13.5 Terceiro algoritmo de Boyer–Moore

O terceiro algoritmo de Boyer–Moore é uma fusão dos dois anteriores: a cada passo, o algoritmo escolhe o maior dos deslocamentos ditados pelas tabelas `T1` e `T2`. Infelizmente, mesmo este algoritmo consome mn unidades de tempo no pior caso. No caso médio, entretanto, ele é muito rápido e consome apenas n unidades de tempo.

A definição da tabela `T2` pode ser aperfeiçoada de tal maneira que o terceiro algoritmo consuma apenas $m + n$ unidades de tempo, mesmo no pior caso.

Exercícios

13.5.1 Escreva o código do terceiro algoritmo de Boyer–Moore.

13.5.2 PROJETO DE PROGRAMAÇÃO. Implemente e teste o terceiro algoritmo de Boyer–Moore. Faça uma versão que produza uma lista de todas as ocorrências da palavra no texto. (Para a fase de testes, escreva uma função que confira o resultado do algoritmo de Boyer–Moore comparando-o com o resultado do algoritmo trivial.) Compare, na prática, o desempenho de sua implementação com o do algoritmo trivial. Invente pares palavra/texto interessantes para fazer os testes.

13.5.3 DIFÍCIL. Investigue as alterações que devem ser feitas na definição da tabela `T2` para que o terceiro algoritmo de Boyer–Moore faça apenas $3n$ comparações entre elementos da palavra e do texto.

13.5.4 Leia o verbete *Boyer–Moore string search algorithm* na Wikipedia [21]. Leia o verbete *String searching algorithm* na Wikipedia.

13.5.5 Veja a página de Charras e Lecroq [4], que contém animações de diversos algoritmos de busca de palavra em texto.

Capítulo 14

Árvores binárias

As árvores da computação têm a tendência de crescer para baixo:
a raiz fica no ar enquanto as folhas se enterram no chão.

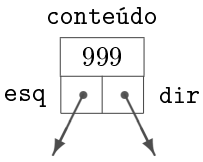
— folclore

Uma árvore binária é uma estrutura de dados mais geral que uma lista encadeada. Este capítulo introduz as operações mais simples sobre árvores binárias. O capítulo seguinte trata de uma aplicação básica.

14.1 Definição

É fácil transmitir a ideia intuitiva de árvore binária por meio de uma figura (veja Figura 14.1), mas é surpreendentemente difícil dar uma definição precisa do conceito. Uma árvore binária é um conjunto de registros (veja Apêndice E) que satisfaz certas condições, detalhadas adiante. Os registros serão chamados **nós** (poderiam também ser chamados **células**). Suporemos, por enquanto, que cada nó tem três campos: um número inteiro e dois ponteiros (veja Apêndice D) para nós. Os nós podem, então, ser definidos assim:

```
struct cel {
   int         conteúdo;
   struct cel *esq;
   struct cel *dir;
};
typedef struct cel nó;
```



O campo `conteúdo` é a "carga útil" do nó, enquanto os outros dois campos dão estrutura à árvore. O campo `esq` contém o endereço de um nó ou `NULL`. Hipótese análoga vale para o campo `dir`. Se o campo `esq` de um nó `X` é o endereço de um nó `Y`, diremos que `Y` é o **filho esquerdo** de `X`. Se `X.esq` = `NULL`, então `X` não tem filho esquerdo. Se `X.dir` = `&Y`, diremos que `Y` é o **filho direito** de `X`. Se `Y` é filho (esquerdo ou direito) de `X`, então `X` é **pai** de `Y`. Uma **folha** é um nó que não tem filho algum.

Um **ciclo** é qualquer sequência $(\mathtt{X}_0, \mathtt{X}_1, \ldots, \mathtt{X}_k)$ de nós tal que $\mathtt{X}_{i+1}$ é filho de $\mathtt{X}_i$ para $i = 0, 1, \ldots, k-1$ e $\mathtt{X}_0$ é filho de $\mathtt{X}_k$. Por exemplo, se `X.esq` = `&X` então (`X`) é um ciclo. Se `X.esq` = `&Y` e `Y.dir` = `&X` então (`X`, `Y`) é um ciclo.

Podemos agora definir o conceito central do capítulo. Uma **árvore binária** é um conjunto $\mathcal{A}$ de nós tal que (1) os filhos de cada elemento de $\mathcal{A}$ pertencem a $\mathcal{A}$, (2) todo elemento de $\mathcal{A}$ tem no máximo um pai, (3) um e apenas um dos elementos de $\mathcal{A}$ não tem pai em $\mathcal{A}$, (4) os filhos esquerdo e direito de cada elemento de $\mathcal{A}$ são distintos e (5) não há ciclos em $\mathcal{A}$. (Em geral, o programador não tem consciência dos detalhes dessa definição porque as árvores são construídas nó a nó de modo a satisfazer as condições naturalmente.) O único elemento de $\mathcal{A}$ que não tem pai em $\mathcal{A}$ é chamado **raiz** da árvore.

Suponha, por exemplo, que `P`, `X`, `Y` e `Z` são nós distintos, que `X` é filho esquerdo de `P`, que `Y` é filho esquerdo de `X`, que `Z` é filho direito de `X` e que `Y` e `Z` são folhas. Então o conjunto {`P`, `X`, `Y`, `Z`} é uma árvore binária. O conjunto {`X`, `Y`, `Z`} também é uma árvore binária.

Subárvores. Um **caminho** em uma árvore binária é qualquer sequência $(\mathtt{Y}_0, \mathtt{Y}_1, \ldots, \mathtt{Y}_k)$ de nós da árvore tal que $\mathtt{Y}_{i+1}$ é filho de $\mathtt{Y}_i$ para $i = 0, 1, \ldots, k-1$. Dizemos que $\mathtt{Y}_0$ é a **origem**, $\mathtt{Y}_k$ o **término** e k o **comprimento** do caminho. Um nó `Z` é **descendente** de um nó `X` se existe um caminho com origem `X` e término `Z`.

Para todo nó `X` de uma árvore binária, o conjunto formado por `X` e todos os seus descendentes é uma árvore binária. Dizemos que esta é a **subárvore** com raiz `X`. Se `P` é um nó, então `P.esq` é a raiz da **subárvore esquerda** de `P` e `P.dir` é a raiz da **subárvore direita** de `P`.

Endereço de uma árvore. O **endereço** de uma árvore binária é o endereço de sua raiz. (O endereço da árvore vazia é `NULL`.) Em discussões informais, é conveniente confundir árvores com seus endereços. Assim, se `r` é o endereço de uma árvore, podemos dizer "`r` é uma árvore" e "considere a árvore `r`". Isso sugere a introdução do nome alternativo `árvore` para o tipo de dados ponteiro–

para–nó:

```
typedef nó *árvore;
```

Recursão. A seguinte observação coloca em evidência a natureza recursiva das árvores binárias. Para toda árvore binária `r`, vale uma das seguintes alternativas:

1. `r` é `NULL` ou
2. `r->esq` e `r->dir` são árvores binárias.

Muitos algoritmos sobre árvores ficam mais simples quando escritos em estilo recursivo.

Exercícios

14.1.1 Dado o endereço `x` de um nó em uma árvore binária, considere a sequência de endereços que se obtém pela iteração das atribuições `x = x->esq` e `x = x->dir` em qualquer ordem. Mostre que esta sequência descreve um caminho.

14.1.2 Mostre que os nós de qualquer caminho em uma árvore binária são distintos dois a dois.

14.1.3 Sejam `X` e `Z` dois nós de uma árvore binária. Mostre que existe no máximo um caminho com origem `X` e término `Z`.

14.1.4 Sequências de parênteses. Árvores binárias têm uma relação muito íntima com certas sequências bem-formadas de parênteses (veja Seção 6.2). Discuta essa relação.

14.1.5 Expressões aritméticas. Árvores binárias podem ser usadas, de maneira muito natural, para representar expressões aritméticas (como $((a+b)*c-d)/(e-f)+g$, por exemplo). Discuta os detalhes desta representação.

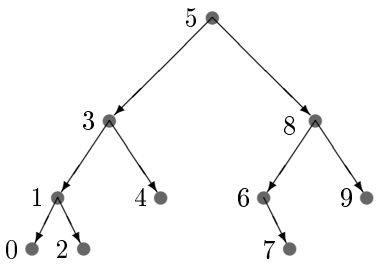


Figura 14.1: Uma árvore binária. Os nós da árvore estão numerados em ordem e-r-d.

14.2 Varredura esquerda-raiz-direita

Os nós de uma árvore binária podem ser visitados em muitas ordens diferentes. Cada ordem define uma **varredura** da árvore. Na varredura **e-r-d**, ou **esquerda-raiz-direita** (*inorder traversal*), visitamos

1. a subárvore esquerda da raiz, em ordem e-r-d,
2. depois a raiz,
3. depois a subárvore direita da raiz, em ordem e-r-d.

Eis uma função recursiva que faz a varredura e-r-d de uma árvore:

```
/* Recebe uma árvore binária r e imprime o conteúdo
 * de seus nós em ordem e-r-d. */
void Erd (árvore r) {
   if (r != NULL) {
      Erd (r->esq);
      printf ("%d\n", r->conteúdo);
      Erd (r->dir);
   }
}
```

A versão iterativa da função `Erd` usa uma pilha (veja Capítulo 6) de nós. A pilha é armazenada num vetor `p[0..t-1]` e há sempre um nó `x` pronto para ser colocado na pilha. A sequência de nós `p[0]`, `p[1]`, ..., `p[t-1]`, `x` é um roteiro do que ainda precisa ser feito: `x` representa a instrução "imprima a subárvore `x`" e cada `p[i]` representa a instrução "imprima o nó `p[i]` e em seguida a subárvore direita de `p[i]`".

```
/* Recebe uma árvore binária r e imprime o conteúdo de
 * seus nós em ordem e-r-d. Supõe que
 * a árvore não tem mais que 100 nós. */
void ErdI (árvore r) {
   nó *p[100], *x;
   int t = 0;
   x = r;
   while (x != NULL || t > 0) {
      /* o topo da pilha p[0..t-1] está em t-1 */
      if (x != NULL) {
         p[t++] = x;
```

```
            x = x->esq;
         }
         else {
            x = p[--t];
            printf ("%d\n", x->conteúdo);
            x = x->dir;
         }
      }
   }
```

As varreduras **r-e-d** (raiz-esquerda-direita ou *preorder traversal*) e **e-d-r** (esquerda-direita-raiz ou *postorder traversal*) são definidas por analogia com a varredura e-r-d.

Exercícios

14.2.1 Encontre um nó com conteúdo k em uma árvore binária.

```
                                  5
                                3 5
                              1 3 5
                            0 1 3 5
                          N 0 1 3 5
0                           N 1 3 5
0 1                           2 3 5
0 1                         N 2 3 5
0 1 2                         N 3 5
0 1 2 3                         4 5
0 1 2 3                       N 4 5
0 1 2 3 4                       N 5
0 1 2 3 4 5                       8
0 1 2 3 4 5                     6 8
0 1 2 3 4 5                   N 6 8
0 1 2 3 4 5 6                   7 8
0 1 2 3 4 5 6                 N 7 8
0 1 2 3 4 5 6 7                 N 8
0 1 2 3 4 5 6 7 8                 9
0 1 2 3 4 5 6 7 8               N 9
0 1 2 3 4 5 6 7 8 9               N
```

Figura 14.2: Função `ErdI` aplicada à árvore binária da Figura 14.1. Para simplificar, confundimos o conteúdo de cada nó com o seu endereço. Cada linha da tabela resume o estado de coisas no início de uma iteração: à esquerda estão os nós que já foram impressos; à direita está a pilha `x`, `p[t-1]`, ..., `p[0]`. A letra `N` representa `NULL`.

14.2.2 Calcule o número de nós de uma árvore binária.

14.2.3 Imprima as folhas de uma árvore binária em ordem e-r-d.

14.2.4 Verifique que o código abaixo é equivalente ao da função `ErdI`:

```
while (1) {
   while (x != NULL) {
      p[t++] = x;
      x = x->esq; }
   if (t == 0) break;
   x = p[--t];
   printf ("%d\n", x->conteúdo);
   x = x->dir; }
```

14.2.5 Escreva uma função que faça a varredura r-e-d de uma árvore binária. Escreva uma função que faça a varredura e-d-r de uma árvore binária.

14.2.6 Escreva uma função que receba uma árvore binária não vazia e devolva o endereço do primeiro nó da árvore na ordem e-r-d. Faça duas versões: uma iterativa e uma recursiva. Repita o exercício com "último" no lugar de "primeiro".

14.2.7 Expressões aritméticas. Discuta a relação entre a varredura e-r-d e a notação infixa de expressões aritméticas. Discuta a relação entre a varredura e-d-r e a notação posfixa. (Veja Seção 6.3 e Exercício 14.1.5.)

14.3 Altura

A **altura de um nó** em uma árvore binária é a distância entre o nó e o seu descendente mais afastado. Mais precisamente, a altura de um nó é o comprimento do mais longo caminho que leva do nó até uma folha.

A **altura de uma árvore** é a altura de sua raiz. Por exemplo, uma árvore

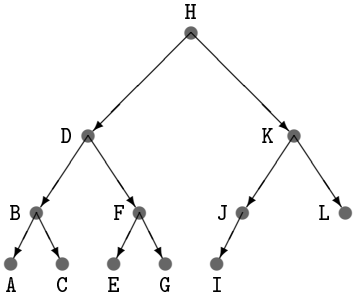


Figura 14.3: Árvore binária quase completa. (A ordem alfabética dos nós descreve uma varredura e-r-d.) A altura da árvore é $\lfloor \log_2 12 \rfloor$.

com um único nó tem altura 0 e a árvore da Figura 14.3 tem altura 3. A altura de uma árvore binária com n nós fica entre $\log_2 n$ e n: se h é a altura da árvore então

$$\lfloor \log_2 n \rfloor \;\leq\; h \;<\; n\,.$$

Uma árvore binária de altura $n-1$ é um "tronco sem galhos": cada nó tem no máximo um filho. Uma árvore binária de altura $\lfloor \log_2 n \rfloor$ é "completa" ou "quase completa": todos os "níveis" estão lotados exceto talvez o último. (Veja Exercício 1.2.4.)

Eis como a altura de uma árvore binária pode ser calculada:

```
/* Devolve a altura da árvore binária r. */
int Altura (árvore r) {
   if (r == NULL)
      return -1; /* a altura de uma árvore vazia é -1 */
   else {
      int he = Altura (r->esq);
      int hd = Altura (r->dir);
      if (he < hd) return hd + 1;
      else return he + 1;
   }
}
```

Árvores balanceadas. Uma árvore binária é **balanceada** se as subárvores esquerda e direita de cada nó tiverem aproximadamente a mesma altura. Uma árvore binária balanceada com n nós tem altura próxima de $\log_2 n$.

Muitos algoritmos sobre árvores binárias consomem tempo proporcional à altura da árvore. Por isso, convém trabalhar com árvores balanceadas. Mas é difícil manter o balanceamento se a árvore sofre inserção e remoção de nós ao longo da execução do algoritmo.

Exercícios

14.3.1 Desenhe uma árvore binária com 17 nós que tenha a menor altura possível.

14.3.2 Escreva uma função iterativa que calcule a altura de uma árvore binária.

14.3.3 Árvores AVL. Uma árvore é balanceada *no sentido AVL* se, para cada nó `x`, as alturas das subárvores esquerda e direita de `x` diferem em no máximo uma unidade. Escreva uma função que decida se uma dada árvore é balanceada no sentido AVL. Procure escrever sua função de modo que ela visite cada nó no máximo uma vez.

14.4 Nós com campo pai

Em algumas aplicações (veja seção seguinte, por exemplo) é conveniente ter acesso imediato ao pai de qualquer nó. Para isso, é preciso acrescentar um campo `pai` a cada nó:

```
struct cel {
   int         conteúdo;
   struct cel *pai;
   struct cel *esq;
   struct cel *dir;
};
typedef struct cel nó;
```

É um bom exercício escrever uma função que preencha o campo `pai` de todos os nós de uma árvore binária.

Exercícios

14.4.1 Escreva uma função que preencha corretamente todos os campos `pai` de uma árvore binária.

14.4.2 A **profundidade** de um nó em uma árvore binária é a distância entre o nó e a raiz da árvore. Mais precisamente, a profundidade de um nó `X` é o comprimento do (único) caminho que vai da raiz até `X`. Por exemplo, a profundidade da raiz é 0 e a profundidade de qualquer filho da raiz é 1. Escreva uma função que determine a profundidade de um nó dado.

14.4.3 É verdade que uma árvore binária é balanceada se e somente se todas as suas folhas têm aproximadamente a mesma profundidade?

14.4.4 Escreva uma função que imprima o conteúdo de cada nó de uma árvore binária precedido de um recuo em relação à margem esquerda do papel. Esse recuo deve ser proporcional à profundidade do nó. Veja Figura 14.4.

14.4.5 HEAP. Em que condições uma árvore binária pode ser considerada um heap (veja Seção 10.1)? Escreva uma função que transforme um max-heap em uma árvore binária quase completa. Escreva uma versão da função `SacodeHeap` (Seção 10.3) para um max-heap representado por uma árvore binária.

14.5 Nó seguinte

Suponha que `x` é o endereço de um nó de uma árvore binária. Queremos calcular o endereço do nó seguinte na ordem e-r-d. Para resolver o problema, é necessário

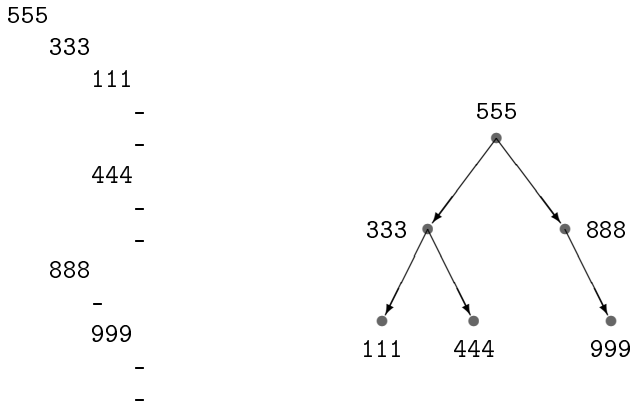


Figura 14.4: O lado esquerdo da figura é uma representação da árvore binária que está à direita. O número de espaços que precede o conteúdo de cada nó é proporcional à profundidade do nó. Os caracteres `'-'` representam `NULL`. Veja Exercício 14.4.4.

que os nós tenham um campo `pai`, conforme a seção anterior. A função abaixo devolve o endereço do nó seguinte a `x` ou devolve `NULL` se `x` é o último nó.

(Às vezes convém confundir, a título de atalho verbal, um nó com o seu endereço. Na documentação da função abaixo, por exemplo, a expressão "recebe um nó `x`" deve ser entendida como "recebe o endereço `x` de um nó". Analogamente, a expressão "devolve o nó seguinte" deve ser entendida como "devolve o endereço do nó seguinte".)

```
/* Recebe um nó x de uma árvore binária cujos nós têm
 * campo pai e devolve o nó seguinte na ordem e-r-d.
 * A função supõe que x != NULL. */
nó *Seguinte (nó *x) {
   if (x->dir != NULL) {
      nó *y = x->dir;
      while (y->esq != NULL) y = y->esq;
      return y; /* 1 */
   }
   while (x->pai != NULL && x->pai->dir == x) /* 2 */[1]
      x = x->pai; /* 3 */
   return x->pai;
}
```

[1] A expressão `x->pai->dir` equivale a `(x->pai)->dir`, conforme o Seção J.5.

Na linha `1` da função `Seguinte`, `y` é o primeiro nó, na ordem e-r-d, da subárvore direita de `x`. As linhas `2` e `3` fazem com que `x` suba na árvore enquanto for filho direito de alguém.

Exercícios

14.5.1 Escreva uma função que receba um nó `x` de uma árvore binária e encontre o nó anterior a `x` na ordem e-r-d.

14.5.2 Escreva uma função que faça varredura e-r-d de uma árvore binária usando a função `Seguinte` e a função sugerida no Exercício 14.2.6.

14.5.3 Leia o verbete *Binary tree* na Wikipedia [21].

Capítulo 15

Árvores binárias de busca

Assim como as árvores binárias são uma generalização das listas encadeadas, as árvores binárias de busca (ou *search trees*) são uma generalização das listas encadeadas crescentes.

15.1 Definição

Considere uma árvore binária cujos nós têm um campo `chave` de um tipo linearmente ordenado, como `int` ou string, por exemplo. Podemos supor que os nós da árvore têm a seguinte estrutura:

```
struct cel {
   int         chave;
   int         conteúdo;
   struct cel *esq;
   struct cel *dir;
};
typedef struct cel nó;
```

(veja Seção 14.1). Uma árvore binária deste tipo é **de busca** (em relação ao campo `chave`) se cada nó `X` tem a seguinte propriedade: a chave de `X` é

1. maior ou igual à chave de qualquer nó na subárvore esquerda de `X` e
2. menor ou igual à chave de qualquer nó na subárvore direita de `X`.

Em outras palavras, para todo nó `X`, todo nó `E` na subárvore esquerda de `X` e todo nó `D` na subárvore direita de `X`, tem-se

$$\texttt{E.chave} \leq \texttt{X.chave} \leq \texttt{D.chave}\,.$$

Esta propriedade equivale à seguinte: a varredura da árvore em ordem e-r-d (veja Seção 14.2) vê as chaves em ordem crescente.

Examinaremos abaixo os problemas de busca, remoção e inserção em árvores de busca. Para estudar esses problemas, convém definir o tipo de dados `árvore` (conforme Seção 14.1):

```
typedef nó *árvore;
```

Exercícios

15.1.1 Escreva uma função que decida se uma dada árvore binária é ou não é de busca.

15.1.2 Suponha que `X.esq->chave` $\leq$ `X.chave` $\leq$ `X.dir->chave` para cada nó `X` de uma árvore binária.[1] Esta árvore é de busca?

15.2 Busca

Dada uma árvore de busca, queremos encontrar um nó cuja chave tenha um certo valor. Eis uma função recursiva que devolve (o endereço de) um nó cuja chave vale k:

```
/* Recebe k e uma árvore de busca r. Devolve um nó cuja
 * chave é k ou devolve NULL se tal nó não existe. */
nó *Busca (árvore r, int k) {
   if (r == NULL || r->chave == k)
      return r;
   if (r->chave > k)
      return Busca (r->esq, k);
   else
      return Busca (r->dir, k);
}
```

No pior caso, a função consome tempo proporcional à altura da árvore (veja Seção 14.3). Se a árvore for balanceada, o consumo será proporcional a $\log_2 n$, sendo n o número de nós.

Eis uma versão iterativa da função `Busca`:

[1] A expressão "`X.esq->chave`" equivale a "`(X.esq)->chave`", conforme o Seção J.5.

```
while (r != NULL && r->chave != k) {
   if (r->chave > k) r = r->esq;
   else r = r->dir;
}
return r;
```

Exercícios

15.2.1 Escreva uma função que encontre uma chave mínima em uma árvore de busca. Escreva uma função que encontre uma chave máxima.

15.2.2 Suponha que as chaves de nossa árvore de busca são distintas duas a duas. Escreva uma função que receba uma chave k e devolva a chave seguinte na ordem crescente.

15.2.3 Escreva uma função que transforme um vetor crescente em uma árvore de busca balanceada (veja Seção 14.3).

15.2.4 Escreva uma função que transforme uma árvore de busca em um vetor crescente.

15.2.5 BUSCA BINÁRIA. Há uma relação muito íntima entre árvores de busca e o algoritmo de busca binária num vetor (veja Capítulo 7). Qual é, exatamente, esta relação?

15.3 Inserção

Considere o problema de inserir um novo nó em uma árvore de busca de tal maneira que a árvore resultante continue sendo de busca. Podemos supor que o novo nó é criado antes que a função de inserção seja invocada:

```
nó *novo;
novo = malloc (sizeof (nó));
novo->chave = k;
novo->esq = novo->dir = NULL;
```

O novo nó será uma folha da árvore. A raiz da nova árvore será a mesma da árvore original, a menos que a árvore original seja vazia.

```
/* Recebe uma árvore de busca r e uma folha avulsa novo.
 * Insere novo na árvore de modo que a árvore continue
 * sendo de busca e devolve o endereço da nova árvore. */
```

```
árvore Insere (árvore r, nó *novo) {
   nó *f, *p;
   if (r == NULL) return novo;
   f = r;
   while (f != NULL) {
      p = f;
      if (f->chave > novo->chave) f = f->esq;
      else f = f->dir;
   }
   if (p->chave > novo->chave) p->esq = novo;
   else p->dir = novo;
   return r;
}
```

Exercícios

15.3.1 Critique a elegância do código da função `Insere`. Escreva uma versão mais elegante. Sugestão: use um ponteiro–para–ponteiro, ou seja, um objeto do tipo `nó **` ou `árvore *`.

15.3.2 Escreva uma versão recursiva da função `Insere`.

15.4 Remoção

Considere o problema de remover um nó de uma árvore de busca de tal forma que a árvore resultante continue sendo de busca. Convém tratar, em primeiro lugar, da remoção da raiz da árvore. Se a raiz tiver apenas um filho, ele assume o papel de raiz. Senão, basta fazer com que o nó anterior à raiz na ordem e-r-d (veja Exercício 14.5.1) assuma o papel de raiz.

```
/* Recebe uma árvore não vazia r, remove a raiz da árvore
 * e rearranja a árvore de modo que ela continue sendo
 * de busca. Devolve o endereço da nova raiz. */
árvore RemoveRaiz (árvore r) {
   nó *p, *q;
   if (r->esq == NULL) q = r->dir;
```

```
   else {
      p = r; q = r->esq;
      while (q->dir != NULL) {
         p = q; q = q->dir;
      }
      /* q é o nó anterior a r na ordem e-r-d */
      /* p é o pai de q */
      if (p != r) {
         p->dir = q->esq;
         q->esq = r->esq;
      }
      q->dir = r->dir;
   }
   free (r);
   return q;
}
```

Agora podemos tratar do caso em que o nó a ser removido não é a raiz da árvore. Para remover o filho esquerdo de um nó `x`, basta fazer

```
x->esq = RemoveRaiz (x->esq);
```

e para remover o filho direito de `x`, basta fazer

```
x->dir = RemoveRaiz (x->dir);
```

Exercícios

15.4.1 Suponha que nós com chaves $50, 30, 70, 20, 40, 60, 80, 15, 25, 35, 45, 36$ são inseridos, nesta ordem, numa árvore de busca inicialmente vazia. Desenhe a árvore que resulta. Em seguida, remova o nó que tem chave 30 de modo que a árvore continue sendo de busca.

15.4.2 Critique a elegância do código da função `RemoveRaiz`. Tente escrever uma versão mais elegante.

15.4.3 Escreva uma versão recursiva da função `RemoveRaiz`.

15.5 Desempenho dos algoritmos

O consumo de tempo de qualquer dos três algoritmos — busca, inserção e remoção — é, no pior caso, proporcional à altura da árvore. Segue daí que convém

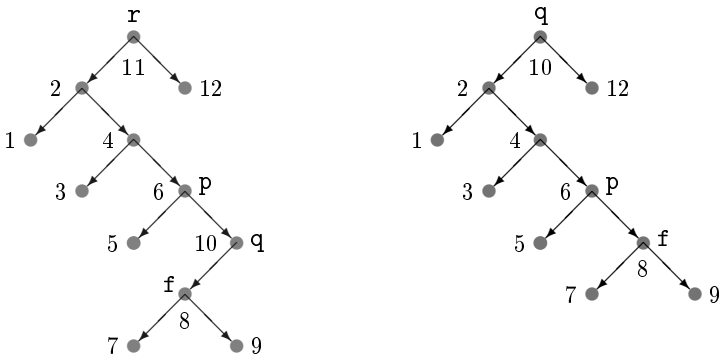


Figura 15.1: Aplicação do algoritmo `RemoveRaiz` à árvore representada à esquerda. (Os nós estão numerados em ordem e-r-d.) O nó `q` é anterior a `r` na ordem e-r-d. O nó `p` é o pai de `q` e o nó `f` é o (único) filho de `q`. O algoritmo faz com que `f` seja o filho direito de `p` e coloca o nó `q` no lugar de `r` (os filhos de `r` passarão a ser os filhos de `q`). A árvore resultante está representada à direita.

trabalhar com árvores balanceadas (veja Seção 14.3). Essas árvores têm altura próxima de $\log_2 n$, sendo n o número de nós.

Infelizmente, os algoritmos de inserção e remoção descritos neste capítulo não produzem árvores balanceadas: se a função `Insere` for repetidamente aplicada a uma árvore balanceada, o resultado pode ser uma árvore bastante desbalanceada; algo análogo pode acontecer depois de uma sequência de invocações da função `RemoveRaiz`. Para enfrentar isso, é preciso inventar algoritmos que façam um rebalanceamento da árvore após cada inserção e cada remoção. Veja, por exemplo, os livros de Sedgewick [18] e Cormen *et al.* [5].

Exercício

15.5.1 Leia o verbete *Binary search tree* na Wikipedia [21].

Apêndice A

Leiaute

"Programming is best regarded as the process of creating works of literature, which are meant to be read."

— D. E. Knuth, *Literate Programming*

"Any fool can write code that a computer can understand. Good programmers write code that humans can understand."

— M. Fowler, *Refactoring: Improving the Design of Existing Code*

Programas precisam ser compreendidos não só por computadores mas também por seres humanos. Embora ignorado pelo computador, o leiaute de um programa — a disposição do código na folha de papel — é muito importante para o leitor humano. Dois aspectos do leiaute são fundamentais:

- os espaços entre as palavras e símbolos em uma linha de código;
- a indentação de cada linha do código (produzida pelos espaços em branco no início da linha).

Embora tratemos aqui do leiaute de programas em linguagem C, as recomendações podem ser aplicadas a muitas outras linguagens de programação.

A.1 Um bom leiaute

Segue uma amostra de bom leiaute. Observe a indentação, o uso correto dos espaços e a posição dos caracteres `{` e `}`.

```
int Funcao (int n, int v[]) {
   int i, j;
   i = 0;
   while (i < n) {
      if (v[i] != 0)
         i = i + 1;
      else {
         for (j = i + 1; j < n; j++)
            v[j-1] = v[j];
         n = n - 1;
      }
   }
   return n;
}
```

Quando as circunstâncias exigem economia de espaço, podemos recorrer ao leiaute compacto abaixo. A indentação deixa clara a estrutura do código sem que o leitor tenha que tropeçar nas chaves { e }.

```
int Funcao (int n, int v[]) {
   int i, j;
   i = 0;
   while (i < n) {
      if (v[i] != 0)  i = i + 1;
      else {
         for (j = i + 1; j < n; j++)  v[j-1] = v[j];
         n = n - 1; } }
   return n; }
```

É fácil habituar-se a produzir um bom leiaute. Com um pouco de prática, os dedos do programador, dançando sobre o teclado, passarão a fazer a coisa certa de maneira autônoma, deixando a mente livre para cuidar de assuntos mais importantes.

Exercício

A.1.1 Critique a tipografia do seguinte código:

```
int Funcao (int n, int v[]) {
  int i, j;
  i = 0;
  while (i < n) {
```

```
    if (v[i] != 0) i = i + 1;
    else {
      for (j = i + 1; j < n; j++) v[j-1] = v[j];
      n = n - 1; } }
  return n; }
```

A.2 Mau exemplo

O leiaute do exemplo abaixo é péssimo. Ele é inconsistente (não faz as coisas sempre da mesma maneira), deixa espaços onde não deve e omite os espaços onde eles são importantes. (Os espaços no código são tão importantes quanto as pausas na música!)

```
int Funcao ( int n,int v[] ){
   int i,j;
   i=0;
   while(i<n){
      if(v[i] !=0)
         i= i +1;
      else
      {
         for (j=i+1;j<n;j++)
            v[j-1]=v[j];
         n =n- 1;
      }
   }
   return n;
}
```

A.3 Sugestões

Para produzir um bom leiaute, não é necessário inventar nada novo. Basta usar as regras tipográficas adotadas por todos os bons livros, revistas e jornais. Eis algumas dessas regras:

1. use um espaço para separar uma palavra da palavra seguinte (os símbolos `=`, `<=`, `while`, `if`, `for` etc. contam como palavras);
2. deixe um espaço depois, mas não antes, de cada sinal de pontuação;
3. deixe um espaço depois, mas não antes, de fechar um parêntese;
4. deixe um espaço antes, mas não depois, de abrir um parêntese.

A expressão "`while(j < n)`", escrita como está, tem o desagradável sabor de "enquantoj for menor que n". Portanto, não escreva

- "`while(j < n)`" no lugar de "`while (j < n)`",
- "`else{`" no lugar de "`else {`",
- "`for (i=1;i<n;i++)`" no lugar de "`for (i = 1; i < n; i++)`",

e assim por diante. Há algumas exceções notórias a essas regras: escreva

- "`x->seg`" e não "`x -> seg`",
- "`x[i]`" e não "`x [i]`",
- "`x++`" e não "`x ++`".

Também é razoável suprimir o espaço entre o nome de uma função e o abre-parêntese seguinte. Por exemplo, é usual escrever "`Funcao(9, v)`" em lugar de "`Funcao (9, v)`". Mas isso não se aplica aos operadores `while`, `for`, `if`, `return`, `sizeof` etc., que *não são funções*.

Exercícios

A.3.1 Para cada um dos pares de linhas abaixo, diga qual das duas linhas tem o melhor leiaute.

```
para j variando de 1 até n de 1 em 1, faça
paraj variando de1 até n de 1em1,faça
```

```
for (j = 0; j < n; j++) {
for(j = 0; j < n; j++) {
```

```
for (j = 0; j < n; j++){
for (j = 0; j < n; j++) {
```

A.3.2 Corrija os erros de leiaute do texto abaixo.

```
Em 1959 e nas décadas seguintes  nenhum programador Cobol
poderia imaginar que os programas de computador que estava
criando ainda estariam em operação no fim do século.Poucos se
lembram hoje de que os primeiros PCs possuíam apenas 64Kbytes
de memória.Como a quantidade de memória disponível era
pequena,usavam-se muitos truques para economizar esse recurso.
Para representar o ano,armazenava-se(por exemplo )"85"em vez
de"1985". com a chegada do ano 2000 , essa codificação
econômica transformou-se em um erro em potencial .
```

A.3.3 Reescreva o código abaixo com leiaute decente.

```
int separa(int v[],int p,int r){int c=v[p],i=p+1,j=r,t;
while(i<=j){if(v[i]<=c){v[i-1]=v[i];++i;}else{t=v[i];
v[i]=v[j];v[j]=t;--j;}}v[j]=c;return j;}
```

A.3.4 Reescreva o fragmento de programa abaixo usando leiaute decente.

```
esq= 0; dir=N-1;
i=(esq+dir)/2;     /*indice do "meio"de R[]*/
while(esq<= dir && R[i] != X){
   if(R[i]<X) esq = i+1;
   else dir = i-1;  /* novo indice do "meio"de R[] */
   i= (esq + dir)/2;
   }
```

A.3.5 Aprenda a usar o programa `indent`, que ajuda a produzir um bom leiaute. O programa está presente em todo sistema UNIX e GNU/Linux.

A.3.6 Leia o verbete *Programming style* na Wikipedia [21].

A.3.7 Familiarize-se com as recomendações do *GNU Coding Standards*, `www.gnu.org/prep/standards/html_node/index.html`.

A.4 Código enfeitado

Neste livro, o código de funções precisa, muitas vezes, ser discutido no texto adjacente. Em tais discussões, é conveniente e apropriado escrever os nomes das variáveis em fonte *itálica*. Se as mesmas variáveis forem escritas em fonte `monoespaçada` dentro do código, teremos margem para confusão. Para evitar esse problema, o livro toma a liberdade, ocasionalmente, de usar fonte itálica dentro do código C. Veja um exemplo:

```
int Função (int n, int v[]) {
   int i, j;
   i = 0;
   while (i < n) {
      if (v[i] != 0)
         i = i + 1;
      else {
         for (j = i + 1; j < n; j++)
            v[j-1] = v[j];
         n = n - 1;
      }
   }
   return n;
}
```

O livro também toma a liberdade de usar letras acentuadas em nomes de funções e de variáveis, embora o alfabeto da linguagem C não tenha tais letras.

Supõe-se que palavras como "`Função`" serão substituídas por "`Funcao`", antes que o código seja submetido ao compilador C.

Exercício

A.4.1 Familiarize-se, ainda que superficialmente, com o sistema CWEB de Knuth e Levy [11]. Trata-se de uma sofisticada ferramenta que ajuda a escrever código C integrado com a correspondente documentação. Amostras de programas escritos em CWEB podem ser vistas em `www.ime.usp.br/~pf/CWEB/exemplo1/tex/mdp.pdf` e `www.ime.usp.br/~pf/CWEB/exemplo2/tex/isort.pdf`.

Apêndice B

Caracteres

Há dois tipos de caracteres em C: sem sinal e com sinal. Um **caractere sem sinal** (ou *unsigned character*) nada mais é que um número do conjunto $0, 1, 2, \ldots, 254, 255$, enquanto um **caractere com sinal** (ou *signed character*) é um número do conjunto $-128, -127, \ldots, 126, 127$. A distinção entre os dois tipos é, em geral, irrelevante. (Os dois tipos são apenas duas interpretações diferentes do conjunto de todas as sequências de 8 bits.) Cada caractere (de qualquer dos dois tipos) é armazenado em um byte (veja Apêndice C) na memória do computador. Para criar uma variável `u` do primeiro tipo, diga

```
unsigned char u;
```

Para criar uma variável `c` do segundo tipo, diga

```
char c;
```

B.1 Representação gráfica dos caracteres

Quando um caractere é exibido na impressora ou na tela do monitor, ele é representado por um símbolo gráfico. No intervalo $0..127$, os dois tipos de caracteres têm os mesmos símbolos gráficos. O símbolo do caractere 65, por exemplo, é

`A`

e o símbolo do caractere 66 é `B`. Alguns caracteres têm símbolos "gráficos" especiais. Por exemplo, o caractere 32 é representado por um espaço, o caractere 0 tem representação vazia (não ocupa espaço algum), e o caractere 10 é representado por uma mudança de linha.

Os símbolos gráficos e os efeitos especiais dos caracteres 0 a 127 foram estabelecidos pelo American Standard Code for Information Interchange. A

32	54 6	76 L	98 b	120 x	199 Ç	231 ç
33 !	55 7	77 M	99 c	121 y	200 È	232 è
34 "	56 8	78 N	100 d	122 z	201 É	233 é
35 #	57 9	79 O	101 e	123 {	202 Ê	234 ê
36 $	58 :	80 P	102 f	124 |	203 Ë	235 ë
37 %	59 ;	81 Q	103 g	125 }	204 Ì	236 ì
38 &	60 <	82 R	104 h	126 ~	205 Í	237 í
39 ’	61 =	83 S	105 i	161 ¡	209 Ñ	241 ñ
40 (	62 >	84 T	106 j	166 ¦	210 Ò	242 ò
41)	63 ?	85 U	107 k	167 §	211 Ó	243 ó
42 *	64 @	86 V	108 l	170 ª	212 Ô	244 ô
43 +	65 A	87 W	109 m	186 º	213 Õ	245 õ
44 ,	66 B	88 X	110 n	188 ¼	214 Ö	246 ö
45 -	67 C	89 Y	111 o	189 ½	217 Ù	249 ù
46 .	68 D	90 Z	112 p	190 ¾	218 Ú	250 ú
47 /	69 E	91 [	113 q	191 ¿	220 Ü	252 ü
48 0	70 F	92 \	114 r	192 À	224 à	255 ÿ
49 1	71 G	93]	115 s	193 Á	225 á	
50 2	72 H	94 ^	116 t	194 Â	226 â	
51 3	73 I	95 _	117 u	195 Ã	227 ã	
52 4	74 J	96 ‘	118 v	196 Ä	228 ä	
53 5	75 K	97 a	119 w	198 Æ	230 æ	

Figura B.1: Amostra da tabela de caracteres ISO 8859-1. A figura mostra os símbolos gráficos da maioria dos caracteres do tipo `unsigned char`. Estão ausentes os caracteres especiais $0, \ldots, 31$ (veja Figura B.3) e diversos outros, irrelevantes no nosso contexto.

correspondência é conhecida como “tabela ASCII”. Os símbolos dos caracteres sem sinal 128 a 255 e os dos caracteres com sinal -128 a -1 não estão bem padronizados: cada sistema escolhe a tabela que mais lhe agrada. Uma das tabelas mais difundidas é a ISO 8859-1, também conhecida como ISO Latin1 (veja Figura B.1). Nesta tabela, cada `char` negativo c tem o mesmo símbolo gráfico que o `unsigned char` $c + 256$ (veja Figura B.2).

Quando os caracteres do tipo `char` são colocados em ordem crescente, as letras acentuadas (como `ã`) precedem as letras não acentuadas (como `a`). O contrário acontece na sequência crescente dos `unsigned char`. Esta é a única diferença relevante entre `unsigned char` e `char`.

Exercícios

B.1.1 Escreva um fragmento de código que receba dois caracteres (sem sinal) via teclado e diga se o primeiro vem antes ou depois do segundo na tabela ISO 8859-1.

B.1.2 Escreva um programa que exiba na tela do monitor os símbolos gráficos dos caracteres (sem sinal) 32 a 255.

`char`	0	1	⋯	127	−128	−127	−126	⋯	−2	−1
`unsigned char`	0	1	⋯	127	128	129	130	⋯	254	255

Figura B.2: Caracteres com sinal e caracteres sem sinal. Os caracteres que estão na mesma coluna da tabela têm o mesmo símbolo gráfico. Cada `char` negativo c tem o mesmo símbolo gráfico que o `unsigned char` $c + 256$.

B.2 Constantes e brancos

Não é cômodo usar expressões como "`97`" e "`-29`" para representar constantes do tipo `char` dentro de um programa C:

```
char c, d, e;
c = 97;  d = -29;  e = 48;
```

É muito mais cômodo escrever o símbolo gráfico do caractere entre aspas simples:

```
c = 'a';  d = 'ã';  e = '0';
```

Quanto aos caracteres que produzem efeitos especiais, usa-se uma representação especial que começa com uma barra invertida. Por exemplo, `'\n'` é o mesmo que `10`, e `'\0'` é o mesmo que `0` (veja Figura B.3).

Os caracteres 9, 10, 11, 12, 13 e 32 são conhecidos como **brancos** (ou *white-spaces*). Muitas funções das bibliotecas da linguagem C tratam todos os brancos como se fossem `' '`. É o caso, por exemplo, da função `scanf` (veja Seção H.1).

Exercícios

B.2.1 Qual a diferença entre `'0'`, `'0'` e `'\0'`?

B.2.2 Interprete os elementos do vetor 70 65 67 73 76 32 67 79 77 79 32 50 43 50 46 como caracteres. Qual o resultado?

B.2.3 Familiarize-se com a função `isspace`, definida na biblioteca `ctype`. Ela recebe um caractere sem sinal c e devolve um inteiro não nulo ou 0 conforme c seja ou não um *white-space*. (Na verdade, o argumento de `isspace` é um `int` cujo valor deve pertencer ao conjunto $0, 1, \ldots, 255$ ou ser igual à constante `EOF` discutida no Seção H.3.)

B.2.4 Familiarize-se com o programa `od` (o nome é uma abreviatura de "*octal dump*"), que recebe um arquivo e imprime o símbolo gráfico e o valor numérico de cada caractere do arquivo. Este utilitário está presente em todo sistema UNIX e GNU/Linux.

`char`	constante C	símbolo gráfico
0	`'\0'`	caractere nulo (*nul*)
9	`'\t'`	tabulação horizontal (*tab*)
10	`'\n'`	mudança de linha (*newline*)
11	`'\v'`	tabulação vertical (*vertical tab*)
12	`'\f'`	quebra de página (*form feed*)
13	`'\r'`	*carriage return*
32	`' '`	espaço
39	`'\''`	'
48	`'0'`	0
65	`'A'`	A
79	`'O'`	O
97	`'a'`	a

Figura B.3: Representação e efeito de algumas constantes do tipo `char`.

B.3 Operações aritméticas

As operações aritméticas envolvendo variáveis do tipo `unsigned char` e `char` são executadas em aritmética `int` (veja Seção C.3) e não em aritmética módulo 256, como alguém poderia supor. Assim, por exemplo, se as variáveis `u` e `v` são do

```
x = 'A'    x = 65    CHAR=A INT=65
x = '0'    x = 48    CHAR=0 INT=48
x = 'ã'    x = -29   CHAR=ã INT=-29
x = '\0'   x = 0     CHAR= INT=0
x = '\n'   x = 10    CHAR=
                      INT=10
```

Figura B.4: Um `char` pode ser convertido num `int` e vice-versa. Em todos os exemplos acima, `x` tanto pode ser uma variável do tipo `char` quanto uma do tipo `int`. Em cada linha, as duas atribuições do lado esquerdo têm exatamente o mesmo efeito. O lado direito mostra o resultado do comando `printf ("CHAR=%c INT=%d", x, x)`, que imprime `x` nos formatos `%c` e `%d`.

tipo `unsigned char` e valem 255 e 2 respectivamente, o valor da expressão `u+v` é 257.

Já as atribuições de um inteiro a um caractere são feitas módulo 256. Assim, se `u` é uma variável do tipo `unsigned char` e `c` é uma variável do tipo `char` então depois de

```
u = 256;
c = 130;
```

o valor de `u` será 0 e o valor de `c` será -126. Por isso, os processos iterativos abaixo "entram em *loop*" (nunca terminam):

```
unsigned char u;
for (u = 0; u < 256; u++) printf (".");
```

```
char c;
for (c = 0; c < 128; c++) printf (".");
```

Já os seguintes fragmentos de código imprimem todas as letras minúsculas, como seria de se esperar:

```
char c;
for (c = 'a'; c <= 'z'; c += 1) printf ("%c\n", c);
```

```
int i;
for (i = 1; i < 26; i++) printf ("%c\n", 'a' + i - 1);
```

Apêndice C

Números: naturais e inteiros

A memória de qualquer computador é uma sequência de bytes. Cada **byte** consiste em 8 bits e portanto tem 256 possíveis valores: `00000000`, `00000001`, ..., `11111110`, `11111111`. Este apêndice procura mostrar como os números naturais (0, 1, 2, 3 etc.) e os números inteiros (positivos e negativos) são representados por sequências de bytes na memória do computador.

O ponto de partida da representação é a notação binária: cada sequência de bits representa o resultado da soma das potências de 2 que correspondem aos bits `1`. Por exemplo, a sequência `010011` representa o número 19, pois $\texttt{0}\times 2^5 + \texttt{1}\times 2^4 + \texttt{0}\times 2^3 + \texttt{0}\times 2^2 + \texttt{1}\times 2^1 + \texttt{1}\times 2^0 = 19$. O conjunto de todas as sequências de k bits representa os números naturais de 0 a $2^k - 1$.

C.1 Representação de números naturais

Na linguagem C, os números naturais são conhecidos como "inteiros sem sinal". Para declarar uma variável `n` deste tipo, diga

```
unsigned int n;
```

Um `unsigned int` é armazenado em s bytes consecutivos, sendo s o valor da expressão `sizeof (unsigned int)`. Portanto, um `unsigned int` é representado por $8s$ bits e assim pode assumir 2^{8s} valores, a saber, $0, 1, \ldots, 2^{8s}-1$. Em alguns computadores, s vale 2 e portanto um `unsigned int` tem 2^{16} valores, de 0 a 65535. Em outros computadores, s vale 4 e assim um `unsigned int` tem 2^{32} valores, que representam os números 0 a 4294967295.

Números naturais maiores que $2^{8s}-1$ são representados módulo 2^{8s}. Se $s = 2$, por exemplo, os números 65536 e 65537 são representados por 0 e 1 respectivamente. Em geral, o programador ignora esse efeito pois trabalha com números muito menores que 2^{8s}.

unsigned int	bits
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
10	1010
11	1011
12	1100
13	1101
14	1110
15	1111

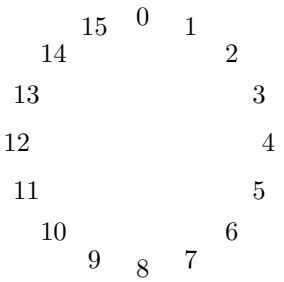


Figura C.1: Imagine um computador em que um `unsigned int` ocupa 1 byte e cada byte tem apenas 4 bits. Os possíveis valores de um `unsigned int` são $0, \ldots, 15$. A correspondência entre estes números e as sequências de 4 bits é dada na tabela. A tabela se fecha num ciclo. Assim, por exemplo, 16 e 17 são representados por 0 e 1.

Exercícios

C.1.1 Qual o efeito do código abaixo se cada `unsigned int` é representado por 16 bits?

```
unsigned int n;
for (n = 0; n < 65536; n++) x[n] = 0;
```

C.1.2 Queremos contar o número de ocorrências de um fenômeno que não acontece mais que 65535 vezes. Podemos usar uma variável do tipo `unsigned int` para a contagem? Suponha agora que o fenômeno ocorre mais que 65535 vezes e proponha um contador em base 100 (ou base 256, ou base 65536) implementado em uma lista encadeada.

C.1.3 Projeto de programação. Escreva um programa que receba um número natural n e imprima as potências n^2, n^3, n^4, n^5 etc. O programa só deve parar quando não for possível armazenar uma potência em um `unsigned int`.

C.2 Representação de números inteiros

Números inteiros (positivos e negativos) são conhecidos em C como "inteiros com sinal". Para criar uma variável `i` deste tipo, basta dizer

```
int i;
```

`unsigned int`	`int`	bits
0	+0	0000
1	+1	0001
2	+2	0010
3	+3	0011
4	+4	0100
5	+5	0101
6	+6	0110
7	+7	0111
8	−8	1000
9	−7	1001
10	−6	1010
11	−5	1011
12	−4	1100
13	−3	1101
14	−2	1110
15	−1	1111

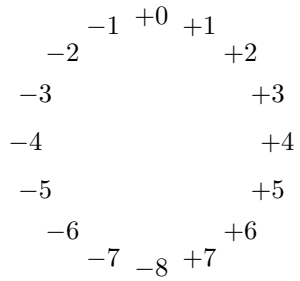


Figura C.2: Imagine que cada `unsigned int` e cada `int` ocupa apenas 1 byte e que cada byte tem apenas 4 bits. Os possíveis valores de um `unsigned int` são $0, 1, \ldots, 15$ e os possíveis valores de um `int` são $-8, -7, \ldots, 6, 7$. A correspondência entre estes números e as sequências de 4 bits é dada na tabela. Os inteiros positivos são representados em notação binária. Os inteiros negativos usam a notação complemento-de-dois: $-i$ é representado pelo inteiro positivo $2^4 - i$ em notação binária. Para efeito das operações aritméticas, a tabela se fecha num ciclo e as operações seguem este ciclo. Por exemplo, se uma variável `i` do tipo `int` vale 6, as expressões `i+1`, `i+2` e `2*i` valem 7, -8 e -4 respectivamente. Nos dois últimos exemplos temos um *overflow* aritmético.

Cada `int` é armazenado em s bytes consecutivos, sendo s o valor da expressão `sizeof(int)` (igual ao valor de `sizeof(unsigned int)`). Assim, cada `int` é representado por uma sequência de $8s$ bits e portanto tem 2^{8s} possíveis valores. O conjunto desses valores é

$$-2^{8s-1}, \ldots, -1, 0, 1, \ldots, 2^{8s-1}-1\,.$$

Cada sequência de $8s$ bits que começa com `0` representa, em notação binária, um `int` não negativo. Cada sequência de $8s$ bits que começa com `1` representa o inteiro negativo $k - 2^{8s}$, sendo k o valor da sequência em notação binária. Esta maneira de representar os inteiros negativos é conhecida como **complemento-de-dois** (ou *two's-complement*).

Números inteiros fora do intervalo $-2^{8s-1} \,..\, 2^{8s-1}-1$ são representados módulo 2^{8s} (ou seja, um número inteiro j é representado pelo único i no intervalo $-2^{8s-1} \,..\, 2^{8s-1}-1$ tal que a diferença $j - i$ é um múltiplo inteiro de 2^{8s}). As atribuições entre `int`s e `unsigned int`s também são feitas módulo 2^{8s}.

Se $s = 2$, por exemplo, o conjunto de valores dos `int`s vai de -2^{15} a $2^{15}-1$, ou seja, de -32768 a 32767. Os números -32769 e 32768 são representados por 32767 e -32768 respectivamente. Depois do fragmento de código

```
unsigned int n;  int i;
n = -1;  i = 65535;
```

o valor de `n` será 65535 e o valor de `i` será -1.

Exercício

C.2.1 Suponha que cada número inteiro é representado por 4 bits. Imagine que cada inteiro negativo de -7 a -1 é representado da seguinte maneira (diferente da convenção complemento-de-dois): primeiro, o valor absoluto do número é representado em binário, da maneira usual; depois, o primeiro bit assume o valor `1`. Por exemplo, -6 é representado por `1110`. Discuta as desvantagens desta notação.

C.3 Aritmética int

As operações aritméticas envolvendo `int`s, `unsigned int`s, `char`s e `unsigned char`s são executadas como se seus operandos fossem do tipo `int` e produzem resultados do tipo `int`. Em particular, os cálculos são feitos módulo 2^{8s}. Por exemplo, se `m` e `n` são variáveis do tipo `unsigned int` e valem 2 e 3 respectivamente, então a expressão `m-n` tem valor -1. Outro exemplo: se uma variável `i` do tipo `int` vale 32767 e s vale 2 então o valor da expressão `i+1` é -32768.

Se o resultado de uma operação precisa ser reduzido módulo 2^{8s} para ficar no intervalo $-2^{8s-1} \,..\, 2^{8s-1}-1$, dizemos que ocorreu um *overflow* aritmético. Em geral, o programador ignora a possibilidade de *overflow* pois trabalha com números de valor absoluto pequeno.

O resultado das divisões é truncado. A expressão `9/2`, por exemplo, tem valor $\lfloor 9/2 \rfloor$. No caso de números negativos, a divisão não obedece o operador piso: na maioria dos computadores, a expressão `-9/2` tem valor $-\lfloor 9/2 \rfloor$, que é diferente de $\lfloor -9/2 \rfloor$.

C.4 Representação por sequências de caracteres

Números inteiros também podem ser representados por sequências de caracteres. O inteiro -123, por exemplo, pode ser representado pela sequência de caracteres

(veja Apêndice B). De acordo com a tabela ISO 8859-1 (veja Seção B.1), esta sequência de caracteres nada mais é que a sequência

45 49 50 51

de números. Como cada caractere é armazenado em um byte de 8 bits, o resultado é a sequência

```
00101101 00110001 00110010 00110011
```

de quatro bytes. Para contrastar esta representação com a discutida no Seção C.2, observe que a representação de -123 em notação complemento-de-dois é

```
11111111 10000101
```

(supondo que cada `int` é armazenado em 2 bytes).

Para converter a representação por sequência de caracteres em representação complemento-de-dois, usa-se a função `atoi` (abreviatura de *alphanumeric to integer*), que está na biblioteca `stdlib` (veja Seção K.1). A função recebe uma string (veja Apêndice G) e devolve o `int` correspondente. Por exemplo, `atoi("9876")` vale 9876 e `atoi("-9876")` vale -9876.

A representação de inteiros por sequências de caracteres não é muito econômica, pois exige muitos bytes. Além disso, é difícil fazer operações aritméticas diretamente sobre esta representação. Por isso, a representação não é usada na memória do computador. Entretanto, a representação é muito usada em arquivos (veja Seção H.2). Se um arquivo usa esta maneira de representar inteiros, dizemos que é um "arquivo-texto" (ou *text file*); se usa a representação complemento-de-dois, dizemos que é um "arquivo binário" (ou *binary file*).

Apêndice D

Endereços e ponteiros

Os conceitos de endereço e ponteiro são fundamentais em qualquer linguagem de programação, embora sejam mais visíveis em C que em outras linguagens. O conceito de ponteiro é difícil; para dominá-lo, é preciso fazer um certo esforço.

D.1 Endereços

A memória de qualquer computador é uma sequência de bytes. Cada byte armazena um de 256 possíveis valores. Os bytes são numerados sequencialmente e o número de um byte é o seu **endereço**.

Cada objeto na memória do computador ocupa um certo número de bytes consecutivos. No meu computador, um `char` ocupa 1 byte, um `int` ocupa 4 bytes e um `double` ocupa 8 bytes. Cada objeto na memória do computador tem um **endereço**. (Na maioria dos computadores, o endereço de um objeto é o endereço do seu primeiro byte.)

Em C, o endereço de um objeto é dado pelo operador `&`. (Não confunda este `&` com o operador lógico *and*, representado por `&&`.) Assim, se `i` é uma

`char c;`	`c`	*89421*
`int i;`	`i`	*89422*
`struct {`	`ponto`	*89426*
`  int x, y;`	`v[0]`	*89434*
`} ponto;`	`v[1]`	*89438*
`int v[4];`	`v[2]`	*89442*

Figura D.1: O lado direito da figura dá os endereços das variáveis declaradas do lado esquerdo. Portanto, `&i` vale *89422* e `&v[3]` vale *89446*. (Os endereços não são muito realistas...)

variável então `&i` é o seu endereço.

O operador `&` aparece frequentemente nas invocações da função `scanf` (veja Seção H.1), por exemplo. Se `i` é uma variável inteira, podemos escrever `scanf ("%d", &i)`.

Figura D.2: Um ponteiro `p`, armazenado no endereço *60001*, contém o endereço de um inteiro. Neste exemplo, `p` vale 89422 enquanto `&p` vale 60001 e `*p` vale −9999. A parte direita da figura é uma representação esquemática muito útil da parte esquerda.

D.2 Ponteiros

Um **ponteiro** (ou **apontador**) é um tipo especial de variável destinado a armazenar endereços. Todo ponteiro pode ter o valor

`NULL`

que é um endereço "inválido". A constante `NULL` está definida no arquivo-interface `stdlib` (veja Seção K.1) e seu valor é *0* na maioria dos computadores.

Se um ponteiro `p` armazena o endereço de uma variável `i`, podemos dizer "`p` aponta para `i`" ou "`p` é o endereço de `i`". Se um ponteiro `p` tem valor diferente de `NULL` então

`*p`

é o valor do objeto apontado por `p`. (Não confunda esse uso de `*` com o operador de multiplicação!) Por exemplo, se `i` é uma variável e `p = &i` então escrever "`*p`" é o mesmo que escrever "`i`".

Há vários tipos de ponteiros: ponteiros para caracteres, ponteiros para inteiros, ponteiros para registros, ponteiros para ponteiros para inteiros etc. Para declarar um ponteiro `p` para um inteiro, escreva[1]

```
int *p;
```

[1] O leiaute das declarações de ponteiros é sabidamente desconfortável. Conceitualmente, um ponteiro–para–int é um novo tipo de dados, o que sugere que se escreva "`int* p`". Do ponto de vista técnico, entretanto, "`*`" modifica a nova variável e não o "`int`". Isto sugere que se escreva "`int *p`". O compilador C aceita qualquer das formas. Também aceita "`int * p`".

Para declarar um ponteiro `q` para um registro `cel`, escreva

```
struct cel *q;
```

```
int x, i = 888;
int *p;          /* p é um ponteiro para um inteiro */
p = &i;          /* p aponta para i */
x = *p + 999;
```

Figura D.3: Exemplo de ponteiro. O código mostra um jeito bobo de fazer "`x = i+999`".

Exercícios

D.2.1 Se `i` é uma variável do tipo `int`, que sentido fazem as expressões `*&i` e `&&i`?

D.2.2 Leia o verbete *Pointer* na Wikipedia [21].

D.3 Uma aplicação

Suponha que precisamos de uma função que troque os valores de duas variáveis inteiras `i` e `j`. A função

```
void troca (int i, int j) {  /* errado! */
   int temp;
   temp = i; i = j; j = temp;
}
```

não produz o efeito desejado, pois recebe apenas *os valores* das variáveis e não as variáveis propriamente ditas. Para obter o efeito desejado, é preciso passar à função os *endereços* das variáveis:

```
void troca (int *p, int *q) {
   int temp;
   temp = *p; *p = *q; *q = temp;
}
```

Para aplicar a função às variáveis `i` e `j` basta dizer

```
troca (&i, &j);
```

Exercícios

D.3.1 Por que o código abaixo está errado?

```
void troca (int *i, int *j) {
   int *temp;
   *temp = *i; *i = *j; *j = *temp; }
```

D.3.2 Um ponteiro pode ser usado para dizer a uma função onde ela deve depositar o resultado de seus cálculos. Escreva uma função `hm` que converta minutos em horas–e–minutos. A função recebe um inteiro t e os endereços de duas variáveis inteiras, digamos h e m, e atribui valores não negativos a essas variáveis de modo que tenhamos $m < 60$ e $60\,h + m = t$. Escreva também uma função `main` que use a função `hm`.

D.3.3 Escreva uma função `mm` que receba um vetor inteiro `v[0..n-1]` e os endereços de duas variáveis inteiras, digamos `min` e `max`, e deposite nestas variáveis o valor de um elemento mínimo e o valor de um elemento máximo do vetor. Escreva também uma função `main` que use a função `mm`.

D.4 Vetores e endereços

Os elementos de qualquer vetor são alocados consecutivamente na memória do computador. Se cada elemento ocupa l bytes, a diferença entre os endereços de dois elementos consecutivos será l. Para livrar o programador da preocupação com o valor de l, o compilador C cria a ilusão de que l vale 1 qualquer que seja o tipo dos elementos do vetor. Por exemplo, depois dos comandos

```
int *v;
v = malloc (100 * sizeof (int));
```

o ponteiro `v` aponta o primeiro elemento de um vetor de 100 inteiros. O endereço do segundo elemento do vetor é `v+1` e o endereço do terceiro é `v+2`. Se `i` é uma variável do tipo `int` então `v + i` é o endereço do (`i`+1)-ésimo elemento do vetor.

A propósito, as expressões `v+i` e `&v[i]` têm exatamente o mesmo valor e portanto as atribuições

```
*(v+i) = 987     e     v[i] = 987
```

têm o mesmo efeito. Todas estas considerações também valem se o vetor tiver sido alocado estaticamente, por uma declaração como

```
int v[100];
```

(Mas nesse caso `v` é uma espécie de "ponteiro constante", cujo valor não pode ser alterado.)

```
for (i = 0; i < 100; i++)  scanf ("%d", v + i);
for (i = 0; i < 100; i++)  scanf ("%d", &v[i]);
```

Figura D.4: Qualquer dos dois fragmentos de código pode ser usado para "carregar" o vetor `v[0..99]`.

Exercícios

D.4.1 Seja `v` um vetor de `int`s. Suponha que cada `int` ocupa 8 bytes no seu computador. Se o endereço de `v[0]` é *55000*, qual o valor da expressão `v + 3`?

D.4.2 Se `v` é um vetor, qual a diferença conceitual entre as expressões `v[3]` e `v+3`.

D.4.3 Suponha que o vetor `v` e a variável `k` foram declarados assim:

```
int v[100], k;
```

Descreva, em português, a sequência de operações que deve ser executada para calcular o valor da expressão `&v[k+9]`.

Apêndice E

Registros e structs

Um **registro** é uma coleção de variáveis, possivelmente de tipos diferentes. Cada uma dessas variáveis é um **campo** do registro. Na linguagem C, registros são conhecidos como `struct`s (abreviatura de *structures*).

E.1 Definição e manipulação de structs

O exemplo abaixo declara um registro `x` com três campos inteiros:

```
struct {
   int dia, mês, ano;
} x;
```

É uma boa ideia dar um nome — `dma`, por exemplo — ao *tipo* de registro:

```
struct dma {
   int dia, mês, ano;
};
struct dma x; /* um registro x do tipo dma */
struct dma y; /* um registro y do tipo dma */
```

Use o operador ponto (`.`) para atribuir valores aos campos do registro:

```
x.dia = 31; x.mês = 12; x.ano = 2008;
```

Exercícios

E.1.1 Escreva uma função que receba dois structs do tipo `dma`, cada um representando uma data válida, e devolva o número de dias que decorreram entre as duas datas.

E.1.2 Escreva uma função que receba um número inteiro que representa um intervalo de tempo medido em minutos e devolva o correspondente número de horas e minutos (por exemplo, 131 minutos é o mesmo que 2 horas e 11 minutos). Use uma struct para representar o par (horas, minutos).

E.1.3 Complete o código da função `FimEvento` da Figura E.1.

```
struct dma FimEvento (struct dma datainício, int duração) {
   struct dma datafim;
   datafim.dia = ...
   datafim.mês = ...
   datafim.ano = ...
   return datafim;
}

int main (void) {
   struct dma a, b;
   int d;
   scanf ("%d %d %d", &a.dia, &a.mês, &a.ano);[1]
   scanf ("%d", &d);
   b = FimEvento (a, d);
   printf ("Termina em %d/%d/%d\n", b.dia, b.mês, b.ano);
   return EXIT_SUCCESS;
}
```

Figura E.1: Um exemplo de `struct`. A função `FimEvento` recebe a data de início de um evento e a duração do evento em dias. Ela devolve a data de fim do evento. O código foi omitido porque é um tanto enfadonho (deve levar em conta o número de dias de diferentes meses, os anos bissextos etc.).

E.2 Ponteiros para structs

Cada registro tem um endereço (veja Seção D.1) na memória do computador. (Você pode imaginar que o endereço do registro é o endereço de seu primeiro campo, mas este detalhe é irrelevante.) É muito comum usar um ponteiro para guardar o endereço de um registro. Por exemplo,

[1] A expressão `&a.dia` é interpretada como `&(a.dia)`. Veja Seção J.5.

```
struct dma *p;    /* p é um ponteiro para registros dma */
struct dma x;
p = &x;           /* p aponta para x */
(*p).dia = 31;[2] /* mesmo efeito que x.dia = 31 */
```

A expressão `p->mês` é uma abreviatura muito útil da expressão `(*p).mês`:

```
p->mês = 12;      /* mesmo efeito que (*p).mês = 12 */
p->ano = 2008;
```

Exercícios

E.2.1 Defina um registro `empregado` para armazenar os dados (nome, data de nascimento, número do documento de identidade, data de admissão, salário) de um empregado de sua empresa. Defina um vetor de `empregados` para armazenar todos os empregados de sua empresa.

E.2.2 Um *racional* é qualquer número da forma p/q, sendo p inteiro e q inteiro não nulo. É conveniente representar cada racional por um registro:

```
struct racional { int p, q; };
```

Vamos convencionar que o campo `q` de todo racional é estritamente positivo e que o máximo divisor comum dos campos `p` e `q` é 1. Escreva uma função que receba inteiros a e b e devolva o racional que representa a/b. Escreva uma função que receba um racional x e devolva o racional $-x$. Escreva funções que recebam racionais x e y e devolvam (1) o racional que representa a soma de x e y, (2) o racional que representa o produto de x por y e (3) o racional que representa o quociente de x por y.

[2] A expressão `(*p).dia` é diferente de `*p.dia`, que equivale a `*(p.dia)`. Veja Seção J.5.

Apêndice F

Alocação dinâmica de memória

A alocação estática designa um lugar para cada variável na memória do computador antes que o programa comece a ser executado. As declarações abaixo, por exemplo, alocam memória para duas variáveis simples e um vetor:

```
char c; int i; int v[10];
```

Em certas aplicações, a quantidade de memória necessária só se torna conhecida durante a execução do programa. Para lidar com essa situação é preciso recorrer à alocação *dinâmica* de memória. A alocação dinâmica é administrada pelas funções `malloc` e `free`, que estão na biblioteca `stdlib` (veja Seção K.1). Para usar a biblioteca, é preciso dizer

```
#include <stdlib.h>
```

no início do programa.

F.1 Função malloc

A função `malloc` (abreviatura de *memory allocation*) aloca um bloco de bytes consecutivos na memória e devolve o endereço desse bloco (veja Apêndice D). O número de bytes é especificado no argumento da função. No seguinte fragmento de código, `malloc` aloca 1 byte:

```
char *ptr;
ptr = malloc (1);
scanf ("%c", ptr);
```

O endereço devolvido por `malloc` é do tipo "genérico" `void *`. No exemplo acima, o endereço é armazenado no ponteiro `ptr`, tornando-se assim um ponteiro–para–`char`.

Para alocar um tipo de dados que ocupa vários bytes, é preciso recorrer ao operador `sizeof`, que dá o número de bytes do tipo especificado:

```
typedef struct {
   int dia, mês, ano;
} data;
data *d;
d = malloc (sizeof (data));[1]
d->dia = 31; d->mês = 12; d->ano = 2008;
```

F.2 A memória é finita

Se a memória já estiver toda ocupada, `malloc` não consegue alocar mais espaço e devolve `NULL`. Antes de prosseguir, convém verificar se esse desastre aconteceu. Por exemplo:

```
d = malloc (sizeof (data));
if (d == NULL) {
   printf ("Socorro! Malloc devolveu NULL!\n");
   exit (EXIT_FAILURE);
}
```

Para não cansar o leitor, os fragmentos de código neste livro não verificam se `malloc` devolveu `NULL`. Mas você não deve deixar de fazer a verificação ao escrever os seus próprios programas.

F.3 Função free

As variáveis alocadas estaticamente dentro de uma função deixam de existir quando a execução da função termina. Já as variáveis alocadas dinâmicamente continuam existindo mesmo depois que a execução da função termina. Se for necessário liberar a memória ocupada por essas variáveis, é preciso fazer isso explicitamente.

A função `free` libera a porção de memória alocada por `malloc`. O comando `free (ptr)` informa o sistema de que o bloco de bytes apontado por `ptr` não será mais usado pelo programa. A próxima invocação de `malloc` poderá tomar posse desses bytes.

A função `free` não deve ser aplicada a uma parte apenas do bloco de bytes alocado por `malloc`; aplique a função sempre ao bloco todo.

[1] As aparências enganam: `sizeof` não é uma função.

Convém não deixar ponteiros "soltos" (*dangling pointers*) no seu programa, pois isso pode ser explorado por hackers para atacar o seu computador. Portanto, depois de cada `free (ptr)`, atribua `NULL` a `ptr`:

```
free (ptr);
ptr = NULL;
```

(Atribuir um valor a um ponteiro que se tornou inútil é decididamente deselegante, mas não há como tratar hackers com elegância . . .) Para não cansar o leitor com detalhes repetitivos, o livro não segue esta norma de segurança.

F.4 Alocação de vetores

Eis como um vetor com `n` elementos pode ser alocado (e depois desalocado) durante a execução de um programa:

```
int *v;
int n, i;
scanf ("%d", &n);
v = malloc (n * sizeof (int));
for (i = 0; i < n; i++) scanf ("%d", &v[i]);
for (i = n; i > 0; i--) printf ("%d ", v[i-1]);
free (v);
```

(Convém lembrar que a padronização ANSI da linguagem C não permite escrever "`int v[n];`" se `n` for uma variável.)

Exercícios

F.4.1 Escreva um programa que leia um inteiro positivo `n` seguido de `n` números inteiros e imprima esses `n` números em ordem invertida (primeiro o último, depois o penúltimo etc.). O seu programa não deve impor quaisquer restrições ao valor de `n`.

F.5 Alocação de matrizes

Matrizes bidimensionais são implementadas como vetores de vetores. Uma matriz com `m` linhas e `n` colunas é um vetor cada um de cujos `m` elementos é um vetor de `n` elementos. O seguinte fragmento de código faz a alocação dinâmica de uma tal matriz:

```
int **A;
int i;
A = malloc (m * sizeof (int *));
for (i = 0; i < m; i++)
   A[i] = malloc (n * sizeof (int));
```

O elemento de `A` que está no cruzamento da linha `i` com a coluna `j` é denotado por `A[i][j]`.

Apêndice G

Strings

Considere um vetor de caracteres que contém pelo menos uma ocorrência do caractere nulo (veja Seção B.2). O segmento inicial do vetor que vai até primeira ocorrência do caractere nulo é uma **string**, ou **cadeia de caracteres**. Depois da execução do fragmento de código abaixo, por exemplo, o vetor `s[0..3]` é uma string (a porção `s[4..9]` do vetor não faz parte da string):

```
char *s;
s = malloc (10 * sizeof (char));
s[0] = 'A';
s[1] = 'B';
s[2] = 'C';
s[3] = '\0';
s[4] = 'D';
```

O **comprimento** de uma string é o seu número de caracteres, sem contar o caractere nulo final. O **endereço** de uma string é o endereço do seu primeiro caractere (veja Seção D.4). Em discussões informais, é usual confundir uma string com o seu endereço: a expressão "considere a string `s`" deve ser entendida como "considere a string cujo endereço é `s`".

G.1 Strings constantes

Para especificar uma string constante, basta embrulhar uma sequência de caracteres num par de aspas duplas. O caractere nulo final fica subentendido. Por exemplo, `"ABC"` é uma string constante e o fragmento de código

```
char *s;
s = "ABC";
```

tem essencialmente o mesmo efeito que o fragmento na introdução deste apêndice. O primeiro argumento das funções `printf` e `scanf` é quase sempre uma string constante. Veja, por exemplo, a expressão `printf ("%d\n", k)`.

```
int ContaVogais (char s[]) {
   int i, j, num = 0;
   char *vogais;
   vogais = "aeiouAEIOU";
   for (i = 0; s[i] != '\0'; i++)
      for (j = 0; vogais[j] != '\0'; j++)
         if (vogais[j] == s[i]) {
            num += 1;
            break;
         }
   return num;
}
```

Figura G.1: Esta função conta o número de vogais em uma string `s`.

Exercícios

G.1.1 Qual a diferença entre `"A"` e `'A'`?

G.1.2 Qual a diferença entre `"mnop"` e `"m\nop"`? Qual a diferença entre `"MNOP"`, `"MN\OP"` e `"MNOP"`?

G.1.3 Qual o comprimento da string `"x=%d\n"`?

G.1.4 Escreva uma função que receba um caractere c e devolva uma string que tem c como único elemento.

G.1.5 O que há de errado com o fragmento de código abaixo? (Veja Seção D.4.)

```
char s[10];  s = "ABC";
```

G.1.6 Escreva uma função que receba uma string `s` e inteiros não negativos i e j e devolva a string que corresponde ao segmento `s[`i`..`j`]`. Sua função não deve alocar novo espaço mas pode alterar a string `s` que recebeu.

G.1.7 Escreva uma função que receba uma string e imprima o número de ocorrências de cada caractere na string. Escreva um programa para testar sua função.

G.1.8 Escreva uma função que receba uma string e substitua cada segmento de dois ou mais caracteres `' '` por um só caractere `' '`.

G.1.9 Escreva uma função que receba uma string de `0`s e `1`s, interprete essa string como um número em notação binária (veja Seção C.1) e devolva o valor desse número.

G.1.10 Escreva uma função que imite o comportamento de `atoi`: ao receber uma string que representa um inteiro devolve o valor desse inteiro. Por exemplo, ao receber `"-9876"` devolve −9876.

G.2 A biblioteca string

A biblioteca `string` (não confunda com a biblioteca `strings`) da linguagem C contém várias funções de manipulação de strings. Para ter acesso à biblioteca, o seu programa deve incluir o arquivo-interface `string.h` (veja Seção K.4):

```
#include <string.h>
```

Descrevemos a seguir duas das funções da biblioteca; uma terceira será discutida na seção seguinte. Nessas descrições, adotaremos a definição

```
typedef char *string;
```

ou seja, trataremos strings como um novo tipo de dados (veja Seção J.3).

A função `strlen` (o nome é uma abreviatura de *string length*) recebe uma string e devolve o seu comprimento. O código da função poderia ser escrito assim:

```
unsigned int strlen (string s) {
   int i;
   for (i = 0; s[i] != '\0'; i++) ;
   return i;
}
```

A função `strcpy` (abreviatura de *string copy*) recebe duas strings e copia a segunda (inclusive o caractere nulo final) para o espaço ocupado pela primeira. O conteúdo original da primeira string é perdido. O usuário só deve invocar esta função se souber que o espaço alocado para a primeira string é suficiente para acomodar a cópia da segunda.[1] O código dessa função poderia ser escrito assim:

```
string strcpy (string s, string t) {
   int i;
   for (i = 0; t[i] != '\0'; i++) s[i] = t[i];
   s[i] = '\0';
   return s;
}
```

[1] *Buffer overflow* é uma das origens mais comuns de bugs de segurança!

(A função devolve o endereço da primeira string, mas o usuário geralmente descarta essa informação redundante.) A função `strcpy` pode ser usada para obter um efeito semelhante ao do exemplo que abriu este apêndice:

```
char s[10];
strcpy (s, "ABC");
```

G.3 Ordem lexicográfica e a função strcmp

A **ordem lexicográfica** entre strings é análoga à ordem entre as palavras em um dicionário. Para comparar duas strings `s` e `t`, procure a primeira posição, digamos k, em que as duas strings diferem. Se `s[`k`]` $<$ `t[`k`]` então `s` é **lexicograficamente menor** que `t` e se `s[`k`]` $>$ `t[`k`]` então `s` é **lexicograficamente maior** que `t`. Se k não está definido então `s` e `t` são idênticas ou uma é prefixo próprio da outra; nesse caso, a string mais curta é lexicograficamente menor que a mais longa.

A função `strcmp` (abreviatura de *string compare*) da biblioteca `string` faz a comparação lexicográfica de duas strings. Ela devolve um número negativo se a primeira string for lexicograficamente menor que a segunda, devolve 0 se as duas strings são iguais e devolve um número positivo se a primeira string for lexicograficamente maior que a segunda.

Embora os parâmetros da função sejam do tipo `char *`, a função se comporta como se eles fossem do tipo `unsigned char *` (veja Apêndice B). Assim, por exemplo, `"bb"` é considerada lexicograficamente menor que `"bbã"`, e `"ba"` é considerada lexicograficamente menor que `"bã"`. O código da função poderia ser escrito assim:

```
int strcmp (string s, string t) {
   int i;
   unsigned char usi, uti;
   for (i = 0; s[i] == t[i]; i++)
      if (s[i] == '\0') return 0;
   usi = s[i]; uti = t[i];
   return usi - uti;
}
```

Exercícios

G.3.1 Discuta as diferenças entre os três fragmentos de código a seguir:

```
char a[8], b[8];
```

```
strcpy (a, "abacate");
strcpy (b, "banana");
```

```
char *a, *b;
a = malloc (8); strcpy (a, "abacate");
b = malloc (8); strcpy (b, "banana");
```

```
char *a, *b;
a = "abacate";
b = "banana";
```

G.3.2 O que há de errado com o seguinte fragmento de código?

```
char *b, *a;
a = "abacate"; b = "banana";
if (a < b) printf ("%s vem antes de %s no dicionário", a, b);
else printf ("%s vem depois de %s no dicionário", a, b);
```

G.3.3 O que há de errado com o seguinte fragmento de código?

```
char *a, *b;
a = "abacate"; b = "amora";
if (*a < *b) printf ("%s vem antes de %s no dicionário", a, b);
else printf ("%s vem depois de %s no dicionário", a, b);
```

Apêndice H

Entrada e saída

Este apêndice descreve superficialmente as funções "de entrada" e "de saída" mais importantes da linguagem C. Todas estão na biblioteca `stdio` (veja Seção K.2). Para ter acesso a essa biblioteca, o seu programa deve incluir uma cópia do arquivo-interface `stdio.h`:

```
#include <stdio.h>
```

H.1 Tela e teclado: printf e scanf

A função `printf` (abreviatura de *print formatted*) exibe uma lista de números, caracteres, strings etc. na tela do monitor. O primeiro argumento da função é uma string (veja Apêndice G) que especifica o formato da exibição.

A função `scanf` (abreviatura de *scan format*) lê do teclado uma lista de números, caracteres, strings etc. O primeiro argumento da função é uma string que especifica o formato da lista a ser lida. Os demais argumentos são os endereços (veja Seção D.1) das variáveis onde os valores lidos serão armazenados. A função trata todos os brancos (*white-spaces*, Seção B.2) como se fossem `' '`.

Exercício

H.1.1 Leia os verbetes *Printf* e *Scanf* na Wikipedia [21].

H.2 Arquivos

Um **arquivo** (ou *file*) é uma sequência de bytes que reside no disco (ou outro dispositivo de armazenamento). A estrutura de um arquivo é semelhante à da

```
#include <stdio.h>
#include <stdlib.h>
int main (void) {
   int a, b;
   double media;
   scanf ("%d %d", &a, &b);
   media = (a + b) / 2.0;
   printf ("A média de %d e %d é %f\n", a, b, media);
   return EXIT_SUCCESS;
}
```

Figura H.1: Funções `printf` e `scanf`. Abaixo, o comportamento do programa, supondo que seu nome é `prog`.

```
prompt> prog
222 333
A média de 222 e 333 é 277.500000
prompt>
```

memória do computador, mas os bytes de um arquivo não podem ser endereçados individualmente. O acesso a um arquivo é estritamente sequencial: para chegar ao 5º byte é preciso passar pelo 1º, 2º, 3º e 4º bytes.

Para que um programa possa manipular um arquivo, é preciso associar a ele uma variável do tipo `FILE` (trata-se de uma struct definida no arquivo `stdio.h`). Esta operação de associação é conhecida como "abertura" do arquivo e é executada pela função `fopen` (abreviatura de *file open*). O primeiro argumento de `fopen` é o nome do arquivo e o segundo argumento é `"r"` ou `"w"` para indicar se o arquivo deve ser aberto para leitura (*read*) ou para escrita (*write*). A função devolve o endereço de um `FILE` (ou `NULL`, se não encontra o arquivo especificado). Depois de usar o arquivo, convém "fechá-lo" com a função `fclose` (abreviatura de *file close*).

Para ler um arquivo, usa-se a função `fscanf` (abreviatura de *file scanf*). Tal como `scanf`, a função `fscanf` devolve o número de objetos efetivamente lidos. O programa da Figura H.2 usa isso para detectar o fim do arquivo.

Stdin e stdout. O teclado é o "arquivo padrão de entrada" (*standard input*). Ele está permanente aberto e é representado pela constante `stdin`. Portanto `fscanf (stdin,...)` equivale a `scanf (...)`. Algo análogo acontece com as funções `printf`, `fprintf` e o "arquivo padrão de saída" `stdout`, que representa a tela do monitor.

```
#include <stdio.h>
#include <stdlib.h>
int main (void) {
   int x, n, k;
   double soma;
   FILE *entrada;
   entrada = fopen ("dados.txt", "r");
   if (entrada == NULL) {
      printf ("\nNão encontrei o arquivo!\n");
      exit (EXIT_FAILURE);
   }
   soma = n = 0;
   while (1) {
      k = fscanf (entrada, "%d", &x);
      if (k != 1) break;
      soma += x;
      n += 1;
   }
   fclose (entrada);
   printf ("A média dos números é %f\n", soma/n);
   return EXIT_SUCCESS;
}
```

Figura H.2: Leitura de um arquivo. O arquivo `dados.txt` contém um ou mais números inteiros separados por brancos (veja Seção B.2). O programa calcula a média dos números.

H.3 As funções putc e getc

Uma função de saída de dados mais básica que `printf` é `putc` (o nome é uma abreviatura de *put character*). Cada invocação da função grava um caractere no arquivo especificado. Se `c` é um caractere e `f` aponta um arquivo, `putc (c, f)` grava `c` no arquivo `f`. Por exemplo, `putc ('*', stdout)` exibe um `*` na tela do monitor.

A função correspondente de leitura de caracteres é `getc` (abreviatura de *get character*). Cada invocação da função lê um caractere do arquivo especificado. Por exemplo, `getc (stdin)` lê o próximo caractere do teclado.

Tipo de objeto que getc devolve. Que acontece se `getc` tenta ler o próximo caractere de um arquivo que já acabou (por exemplo, tenta ler um arquivo vazio)? Gostaríamos que `getc` devolvesse algum tipo de "caractere inválido", mas todos os 256 caracteres são "válidos"! Para resolver o impasse, `getc` não devolve um `char` mas sim um `int`, pois o conjunto de valores do tipo `int`

```
#include <stdio.h>
#include <stdlib.h>
int main (void) {
   char linha[100];
   int i, j;
   for (j = 0; ; j++) {
      linha[j] = getc (stdin);
      if (linha[j] == '\n') break;
   }
   for (i = 0; i <= j; i++)
      putc (linha[i], stdout);
   return EXIT_SUCCESS;
}
```

Figura H.3: Uso das funções `getc` e `putc`. O programa lê uma linha de caracteres do teclado, armazena esta linha em um vetor e em seguida exibe os caracteres na tela do monitor.

```
#include <stdio.h>
#include <stdlib.h>
int main (void) {
   char c;  /* erro */
   FILE *entrada;
   entrada = fopen ("dados.txt", "r");
   if (entrada == NULL) exit (EXIT_FAILURE);
   c = getc (entrada);
   fclose (entrada);
   putc (c, stdout);
   return EXIT_SUCCESS;
}
```

Figura H.4: Uso de `getc`. O programa promete ler o primeiro caractere do arquivo `dados.txt` e exibir esse caractere na tela do monitor. Se o arquivo estiver vazio, o programa não funciona corretamente.

```
int c;
c = getc (entrada);
if (c != EOF) putc (c, stdout);
```

Figura H.5: Alterações do programa da Figura H.4. Se o arquivo estiver vazio, o programa não imprime nada.

contém propriamente o conjunto de valores do tipo `char`. Se o arquivo tiver acabado, `getc` devolve um `int` que não possa ser confundido com um `char`. Mais especificamente,

1. se houver um próximo caractere no arquivo, `getc` lê o caractere como se ele fosse um `unsigned char`, transforma-o em um `int` e devolve o resultado;
2. se o arquivo não tiver mais caracteres, `getc` devolve -1.

Mais precisamente, se o arquivo não tiver mais caracteres, a função devolve a constante `EOF` (abreviatura de *end of file*), que está definida no arquivo-interface `stdio.h` (veja Seção K.2). O valor usual de `EOF` é -1.

A solução adotada por `getc` é uma boa lição de projeto de algoritmos: a função devolve um objeto que pertence a um *superconjunto* do conjunto em que estamos realmente interessados. Esse tipo de truque é usado em muitas outras situações.

Exercícios

H.3.1 Suponha que o arquivo `dados.txt` contém a sequência de caracteres `ABCÿDEF` e nada mais. Qual o resultado do seguinte programa? Que acontece se trocarmos a declaração `int c` por `char c` ou por `unsigned char c`?

```
int main (void) {
   int c;
   FILE *entrada;
   entrada = fopen ("dados.txt", "r");
   while ((c = getc (entrada)) != EOF)
      printf ("%c ", c);
   fclose (entrada);
   return EXIT_SUCCESS; }
```

H.3.2 Escreva um programa que faça uma cópia de um arquivo. Os nomes dos dois arquivos são digitados pelo usuário.

H.3.3 Suponha dado um arquivo contendo código C com comentários. Escreva um programa que leia esse arquivo, remova os comentários, e grave o código "limpo" em outro arquivo.

H.4 Argumentos na linha de comando

A função `main`, como qualquer outra função, admite argumentos. Eles são conhecidos como "argumentos na linha de comando" (ou *command-line arguments*). O primeiro argumento é um inteiro que dá o número de argumentos.

O segundo, é um vetor de strings (veja Apêndice G) que armazena os demais argumentos. A função `main` deve ser especificada assim:

```
int main (int numargs, char *arg[]) {
   . . .
}
```

Se o nome do arquivo que contém o programa for `prog` então depois do comando

```
prog a bb   999
```

o parâmetro `numargs` terá valor 4 e os elementos `arg[0]` a `arg[3]` do vetor `arg` serão as strings `"prog"`, `"a"`, `"bb"` e `"999"` respectivamente.

```
#include <stdio.h>
#include <stdlib.h>
int main (int numargs, char *arg[]) {
   int n;
   double soma = 0;
   for (i = 1; i < numargs; i++)
      soma += atoi (arg[i]);
   n = numargs - 1;
   printf ("média = %.2f\n", soma / n);
   return EXIT_SUCCESS;
}
```

Figura H.6: O programa calcula a média dos números fornecidos como argumentos na linha de comando. Abaixo, o comportamento do programa, supondo que seu nome é `prog`.

```
prompt> prog +22 33 -11 +44
média = 22.00
prompt>
```

Exercício

H.4.1 Escreva um programa que conte o número de ocorrências de cada um dos caracteres de um arquivo. O seu programa deve receber o nome do arquivo na linha de comando e imprimir uma tabela com o número de ocorrências de cada caractere do arquivo. Para ganhar inspiração, analise o comportamento do utilitário `wc` (abreviatura de *word count*) presente em todo sistema UNIX e GNU/Linux.

Apêndice I

Números aleatórios

Sequências de números aleatórios são úteis em inúmeras aplicações. Dada a dificuldade de obter números verdadeiramente aleatórios, devemos nos contentar com números *pseudo*-aleatórios, gerados por algoritmos. Para simplificar a linguagem, omitiremos o "pseudo" no que segue.

I.1 A função rand

A função `rand` (abreviatura de *random*), definida na biblioteca `stdlib`, gera números aleatórios no intervalo fechado `0..RAND_MAX`. A constante `RAND_MAX` está definida no arquivo-interface `stdlib.h` (veja Seção K.1). Cada invocação da função produz um número aleatório.

Exercícios

I.1.1 (Roberts [15]) Qual o defeito da seguinte função, que promete simular uma jogada de moeda?

```
char *CaraCoroa (void) {
    int r;
    r = rand () % 2;
    if (r == 1) return "cara";
    else return "coroa"; }
```

I.1.2 O seguinte fragmento de código foi extraído de um programa que pretende simular o rolar de um dado. Qual o defeito do fragmento?

```
if (r < RAND_MAX * 5 / 6) return 5;
```

I.2 Inteiros aleatórios

A função `rand` pode ser usada para produzir um número inteiro aleatório num dado intervalo `a..b`. O código abaixo (adaptada do livro de Roberts [15]) produz o resultado em três passos. Primeiro, transforma o inteiro gerado por `rand` em um número real x tal que $0 \le x < 1$. Depois, transforma x em um número inteiro `i` no intervalo fechado `0..b-a`. Finalmente, translada `i` para o intervalo `a..b`.

```
/* Esta função devolve um inteiro aleatório
 * no intervalo fechado a..b. */
int InteiroAleatório (int a, int b) {
   double r, x, R = RAND_MAX;
   int i;
   r = rand ();
   x = r / (R + 1);
   i = x * (b - a + 1);
   return a + i;
}
```

(A qualidade dos números produzidos pela função não é boa se a diferença `b − a` for grande, especialmente se a diferença for maior que `RAND_MAX`.)

Exercícios

I.2.1 No código da função `InteiroAleatório`, que acontece se escrevermos "`x = r/(RAND_MAX+1)`"? Que acontece se escrevermos "`x = r/RAND_MAX`"?

I.2.2 Verifique que o número produzido pela função `InteiroAleatório` quando `a` = 0 e `b` = `RAND_MAX` é igual ao produzido por `rand`.

I.2.3 Use a função `InteiroAleatório` para simular o rolar de um dado.

I.3 Sementes

Cada vez que é invocada, a função `rand` calcula um novo número aleatório a partir do número, digamos r, que produziu em resposta à invocação anterior. O número r que corresponde à *primeira* invocação de `rand` é conhecido como **semente**. Dada a semente, a sequência de números gerada pelas sucessivas execuções de `rand` está completamente determinada.

O programador pode especificar a semente por meio da função `srand`, que tem um `unsigned int` como argumento. (A função está na biblioteca `stdlib`.) Se o programador não especificar a semente, o sistema adota o valor 0. Para tornar a semente mais imprevisível, pode-se usar a função `time` da biblioteca `time`:

```
srand (time (NULL));
```

Apêndice J

Miscelânea

Este apêndice reúne pequenas notas sobre alguns recursos da linguagem C.

J.1 Valor de uma expressão

Toda expressão em C tem um *valor*. Por exemplo, se `x` vale 99 então a expressão `3 * x + 4` tem valor 301. A expressão

```
y = 3 * x + 4
```

também tem valor 301. O cálculo do valor desta expressão tem o *efeito colateral* de atribuir o valor da subexpressão `3 * x + 4` à variável `y`. Assim, `y` passa a valer 301.

Se `x` vale 99 então o valor da expressão `x += 1` (que é uma abreviatura de `x = x + 1`) é 100. O cálculo do valor desta expressão tem o efeito colateral de incrementar o valor de `x`, que passa a valer 100. Um exemplo mais delicado, muito comum em C: se `x` vale 99 então as expressões

```
x++   e   ++x
```

valem 99 e 100 respectivamente. O cálculo do valor de qualquer destas expressões tem o efeito colateral de incrementar o valor de `x`, que passa a valer 100. Algo análogo vale para expressões da forma `x--` e `--x`.

As expressões que contêm o operador vírgula são usadas, em geral, mais por seu efeito colateral que por seu valor. Por exemplo, o valor da expressão

```
j = 1+2, k = 99
```

é 99 e o cálculo do valor da expressão tem o efeito colateral de atribuir os valores 3 e 99 a `j` e `k` respectivamente. Outro exemplo: as duas expressões abaixo têm o mesmo efeito:

```
t = a, a = b, b = t;
t = a; a = b; b = t;
```

J.2 Valor de uma expressão booleana

Toda expressão booleana tem valor 0 ou 1. Por exemplo, se `j` é igual a 99 e `a` é menor que `b` então o valor da expressão

```
j == 99 && a < b
```

é 1. Caso contrário, ou seja, se `j` $\neq 99$ ou `a` $\geq$ `b`, o valor da expressão é 0.

O valor de toda expressão booleana é calculado *da esquerda para a direita*. Tão logo o valor da expressão fica definido, o cálculo é *interrompido*. Muitas vezes, os últimos termos de uma expressão longa não são sequer examinados. Esta regra (veja *short-circuit evaluation* na Wikipedia [21]) é usada em C para evitar o cálculo de subexpressões de valor indefinido. Por exemplo, embora os fragmentos

```
if (j <= 99 && v[j] < x) ...
if (v[j] < x && j <= 99) ...
```

pareçam equivalentes, o primeiro está correto enquanto o segundo pode estar incorreto se `v[j]` não estiver definido quando `j` é maior que 99.

J.3 Tipos de dados e typedef

Um **tipo de dados** é um conjunto de *valores* munido de um conjunto de *operações*. Algumas operações transformam valores em outros valores. Outras, associam números a valores ou conjuntos de valores. Eis alguns tipos de dados básicos da linguagem C:

1. O conjunto de valores do tipo `int` é `INT_MIN`, $\ldots, -1, 0, 1, \ldots,$ `INT_MAX` (veja Apêndice C). Os valores das constantes `INT_MIN` e `INT_MAX` estão definidos no arquivo-interface `limits.h` (veja Seção K.5). Entre as operações sobre esses valores destacam-se as de comparação (`<`, `<=`, `==`, `>=`, `>`) e as aritméticas (`+`, `-`, `*`, `/`). Cada operação aritmética recebe dois `int`s e devolve um `int` como resultado (veja Seção C.3).
2. O conjunto de valores do tipo `char` (veja Apêndice B) é $-128, -127, \ldots,$ $126, 127$. As operações usuais sobre esses valores são indicadas por `<`, `<=`, `==`, `>=`, `>`, `+` e `-`. Os resultados dessas operações são do tipo `int`.
3. O conjunto de valores do tipo `double` é uma aproximação do conjunto dos

números reais. Os valores mínimo e máximo estão registrados no arquivo-interface `limits.h` (veja Seção K.5). As operações são uma aproximação das operações aritméticas sobre números reais.

O programador pode definir os seus próprios tipos de dados recorrendo ao operador `typedef`. É muito conveniente, por exemplo, definir o tipo de dados `string` (veja Apêndice G):

```
typedef char *string;
```

Para definir um tipo de dados via `typedef`, faça o seguinte: (1) escreva a declaração de uma variável do tipo desejado e (2) escreva "`typedef`" antes da declaração. Isso transformará o nome da variável no nome de um tipo de dados. Por exemplo,

```
struct {double x, y;} ponto_no_plano;
```

declara uma variável `ponto_no_plano`, e

```
typedef struct {double x, y;} ponto_no_plano;
```

transforma "`ponto_no_plano`" em um novo tipo de dados. Esse novo tipo pode ser usado para declarar novas variáveis:

```
ponto_no_plano p, q;
```

J.4 Include e define

Antes de transformar um programa em código executável, o compilador C faz um pré-processamento que cuida de todos os `#include` e `#define`. Cada `#include` é substituído por uma cópia do arquivo indicado. Já a diretiva `#define` faz com que cada ocorrência da constante indicada seja substituída pelo seu valor. Por exemplo, o programa

```
#include <stdlib.h>
#define MAX 1000
int main (void) {
   int v[MAX], i;
   for (i = 0; i < MAX; i++) scanf ("%d", &v[i]);
   return EXIT_SUCCESS;
}
```

é transformado pelo pré-processador no programa

```
int rand (void);
void srand (unsigned int);
int atoi (char *);
int main (void) {
   int v[1000], i;
   for (i = 0; i < 1000; i++) scanf ("%d", &v[i]);
   return 0;
}
```

As primeiras linhas do novo programa são uma cópia do que está no arquivo `stdlib.h`. (O exemplo é fictício: o arquivo `stdlib.h` contém bem mais que essas três linhas. Veja uma amostra maior no Seção K.1.) A constante `MAX` é substituída por `1000` e a constante `EXIT_SUCCESS` é substituída por `0` porque o arquivo `stdlib.h` contém a linha

```
#define EXIT_SUCCESS  0
```

J.5 Precedência entre operadores em C

Há uma relação de precedência entre os vários operadores da linguagem C. Durante o cálculo do valor de uma expressão, alguns operadores são aplicados antes de outros, respeitando a ordem de precedência.

expressão	interpretação	expressão	interpretação
`&x[i]`	`&(x[i])`	`a[i].b[j]`	`((a[i]).b)[j]`
`*p.dia`	`*(p.dia)`	`h->e->d`	`(h->e)->d`
`*x++`	`*(x++)`	`&h->e`	`&(h->e)`

Figura J.1: Alguns exemplos da relação de precedência entre operadores.

Os operadores em cada linha da tabela abaixo têm precedência sobre os operadores das linhas seguintes. A coluna direita dá a regra de associação — esquerda–para–direita ou direita–para–esquerda — dos operadores da linha. Os operadores unários (um só operando) estão todos na segunda linha da tabela; as demais linhas tratam de operadores binários (dois operandos).

binários	`()` `[]` `->` `.`	e–d
unários	`-` `++` `--` `!` `&` `*` `~` (*cast*) `sizeof`	d–e
binários	`*` `/` `%`	e–d
binários	`+` `-`	e–d
binários	`<<` `>>`	e–d
binários	`<` `<=` `>=` `>`	e–d
binários	`==` `!=`	e–d
binário	`&`	e–d
binário	`^`	e–d
binário	`|`	e–d
binário	`&&`	e–d
binário	`||`	e–d
binários	`?` `:`	d–e
binários	`=` *op*`=`	d–e
binário	`,`	e–d

A expressão *op*`=` representa os operadores `+=`, `-=`, `*=` etc. A expressão (*cast*) representa os operadores `(int)`, `(double)` etc.

Apêndice K

Arquivos-interface de bibliotecas

Programas em C têm acesso a várias bibliotecas de funções. A implementação dessas funções varia de computador para computador. Para isolar o usuário das peculiaridades da implementação, o acesso a cada biblioteca se dá através de um arquivo-interface, que contém os protótipos das funções da biblioteca e as definições de algumas constantes. (No meu computador, todos os arquivos-interface estão no diretório `/usr/include/`.)

Para usar uma biblioteca `X`, o programador deve incluir no seu programa (veja Seção J.4) uma cópia do arquivo-interface `X.h`. Para usar a biblioteca `stdlib`, por exemplo, diga

```
#include <stdlib.h>
```

no início do programa. Seguem amostras muito simplificadas (e por vezes um pouco distorcidas) dos arquivos-interface das bilbiotecas `stdlib`, `stdio`, `math`, `string`, `limits` e `time`.

K.1 Amostra do arquivo stdlib.h

```
/* ALOCAÇÃO DE MEMÓRIA, NÚMEROS ALEATÓRIOS ETC.
 ***************************************************************/

/* A função devolve 1 para dizer que terminou de maneira anormal. */

#define EXIT_FAILURE  1

/* Devolve 0 se terminou normalmente. */

#define EXIT_SUCCESS  0
```

```
/* A função exit interrompe a execução do programa e fecha os arquivos
 * que o programa tenha porventura aberto. Se for invocada com
 * argumento 0, o sistema operacional é informado de que o programa
 * terminou com sucesso; caso contrário, o sistema operacional é
 * informado de que o programa terminou de maneira excepcional.
 * Uso típico: exit (EXIT_FAILURE). */

void exit (int status);

#define RAND_MAX  (32767)

/* A função rand devolve um número inteiro pseudoaleatório no
 * intervalo fechado 0..RAND_MAX. Uso típico: i = rand (). */

int rand (void);

/* A função srand define uma semente para a função rand. A função deve
 * ser invocada antes do primeiro uso de rand para que a sequência de
 * números devolvidos por rand não seja sempre a mesma. Uso típico:
 * srand (time (NULL)). */

void srand (unsigned int u);

/* A função atoi recebe uma string que representa um número inteiro em
 * notação decimal e converte essa string no int correspondente.
 * Exemplo: atoi ("-1234") vale -1234. Uso típico: i = atoi (s).
 * É responsabilidade do usuário garantir que o número representado
 * pela string pertence ao intervalo fechado INT_MIN..INT_MAX. */

int atoi (char *s);

#define NULL  0

/* A função malloc recebe um número natural N e aloca N bytes
 * consecutivos na memória. Devolve o endereço do primeiro byte
 * alocado. Se não puder alocar os N bytes, devolve NULL.
 * Uso típico: pntr = malloc (N). */

void *malloc (unsigned int N);

/* A função free recebe o endereço pntr de um bloco de bytes
 * previamente alocado por malloc e libera esse bloco.
 * Uso típico: free (pntr). */

void free (void *pntr);
```

```
/* A função qsort rearranja o vetor base[0..nmemb-1] em ordem
 * crescente. Cada elemento ocupa size bytes. A comparação entre
 * elementos do vetor é dada pela função compar. */

void qsort (void *base, unsigned int nmemb, unsigned int size,
            int (*compar)(void *, void *));
```

K.2 Amostra do arquivo stdio.h

```
/* FUNÇÕES DE ENTRADA E SAÍDA
 ************************************************************************/

#define NULL  0

/* A constante EOF é um inteiro diferente de todo unsigned char. */

#define EOF  (-1)

/* A função fgetc lê o próximo byte (se tal existir) do arquivo f.
 * O byte é interpretado como um unsigned char, convertido para int,
 * e em seguida devolvido pela função. Se f não tem um próximo byte,
 * a função devolve EOF. Uso típico: i = fgetc (f). */

int fgetc (FILE *f);

/* A função getc tem o mesmo comportamento que fgetc mas é
 * implementada como uma macro. Uso típico: i = getc (f). */

int getc (FILE *f);

/* A função fputc converte c em unsigned char e escreve o caractere
 * resultante no arquivo f. Uso típico: fputc (c, f). */

int fputc (int c, FILE *f);

/* A função putc tem o mesmo comportamento que fputc, mas é
 * implementada como uma macro. Uso típico: putc (c, f). */

int putc (int c, FILE *f);

/* Funções de leitura e gravação com formato. */

int printf (char *s, ...);
int scanf (char *s, ...);
int fprintf (FILE *f, char *s, ...);
```

```
int fscanf (FILE *f, char *s, ...);

/* Arquivos. (A definição da struct FILE foi impiedosamente
 * simplificada...) */

typedef struct {
   int            _cnt;
   unsigned char *_ptr;
   unsigned char *_base;
   unsigned char  _flag;
   unsigned char  _file;
} FILE;

extern FILE *stdin;
extern FILE *stdout;

/* A função fopen abre o arquivo cujo nome é str. O arquivo é aberto
 * para leitura se modo for "r" e para gravação se modo for "w".
 * Uso típico: f = fopen ("dir/meuarquivo.txt", "r"). */

FILE *fopen (char *str, char *modo);

/* A função fclose fecha o arquivo f. Uso típico: fclose (f). */

int fclose (FILE *f);
```

K.3 Amostra do arquivo math.h

```
/* PARA TER ACESSO À BIBLIOTECA MATH É PRECISO COMPILAR O PROGRAMA
 * COM AS OPÇÕES APROPRIADAS. NO CASO DO COMPILADOR gcc, DIGA
 *      gcc nome_do_arquivo.c -lm
 ****************************************************************/

/* Funções trigonométricas. */

double sin (double x);
double cos (double x);
double tan (double x);

/* A função exp devolve e^x, ou seja, o número e elevado à potência x.
 * Uso típico: y = exp (x). */

double exp (double x);
```

```
/* A função log devolve o logaritmo de x na base e. Não use com x
 * negativo. Uso típico: y = log (x). */

double log (double x);

/* A função sqrt devolve a raiz quadrada de x. Não use com x negativo.
 * Uso típico: y = sqrt (x). */

double sqrt (double x);
```

K.4 Amostra do arquivo string.h

```
/* FUNÇÕES DE MANIPULAÇÃO DE STRINGS
 ************************************************************************/

/* A função strlen devolve o número de caracteres da string x (sem
 * contar o '\0' final). O código da função tem o mesmo efeito que
 *       for (i = 0; x[i] != 0; i++) ;
 *       return i;
 * Uso típico: k = strlen (x). */

unsigned int strlen (char *x);

/* A função strcmp compara lexicograficamente as strings x e y.
 * Devolve um número negativo se x precede y, devolve 0 se x é igual
 * a y e devolve um número positivo se x sucede y. O código da função
 * equivale a
 *       for (i = 0; x[i] == y[i]; i++)
 *          if (x[i] == 0) return 0;
 *       return (unsigned int) x[i] - (unsigned int) y[i];
 * Uso típico: if (strcmp (x, y) == 0) ... . */

int strcmp (char *x, char *y);

/* A função strcpy copia a string x no espaço alocado para a string y.
 * Antes de invocar a função, certifique-se de que o espaço alocado a
 * y tem pelo menos strlen (x) + 1 bytes. A função devolve y. Exemplo:
 *       char y[4];
 *       strcpy (y, "ABC");
 * O código da função equivale a
 *       for (i = 0; (y[i] = x[i]) != 0; i++) ;
 * Uso típico: strcpy (y, x). */

char *strcpy (char *y, char *x);
```

K.5 Amostra do arquivo limits.h

```
/* VALORES MÍNIMOS E MÁXIMOS DE VÁRIOS TIPOS DE DADOS
 ********************************************************************/

/* Valores mínimo (-2^31) e máximo (2^31-1) do tipo de dados int. */

#define INT_MIN  (-2147483648)
#define INT_MAX  (2147483647)

/* Valor máximo (2^32-1) do tipo de dados unsigned int. */

#define UINT_MAX  (4294967295)

/* Valores mínimo e máximo do tipo de dados double. */

#define DBL_MIN  (2.2250738585072014E-308)
#define DBL_MAX  (1.7976931348623157E+308)
```

K.6 Amostra do arquivo time.h

```
/* MEDIDA DE TEMPO
 ********************************************************************/

/* A função time devolve a leitura do relógio, em segundos. Uso
 * típico: tempo = time (NULL). */

long time (long *t);

#define CLOCKS_PER_SEC  (1000000)

/* A função clock devolve o tempo de CPU decorrido desde o início da
 * execução do seu programa. Para converter essa quantidade de tempo
 * em segundos, divida pela constante CLOCKS_PER_SEC. Exemplo:
 *       double start, finish, elapsed;
 *       start = (double) clock () / CLOCKS_PER_SEC;
 *       . . . [cálculos] . . .
 *       finish = (double) clock () / CLOCKS_PER_SEC;
 *       elapsed = finish - start;
 * Sugestão: repita o bloco [cálculos] muitas vezes, digamos 100,
 * e divida elapsed por esse número. */

long clock (void);
```

Apêndice L

Soluções de alguns exercícios

Este apêndice reúne soluções (nem sempre completas) de alguns dos exercícios.

1.1.3 Tudo indica que o parâmetro `s` é supérfluo.

1.1.4 Tudo indica que a condição "`p <= q <= r`" é supérflua.

2.2.1 A função devolve o valor de um elemento máximo do vetor `v[0..n-1]`:

```
int Máximo (int v[], int n) {
   int j, x = v[0];
   for (j = 1; j < n; j++) {
      if (x < v[j]) x = v[j];
   return x; }
```

2.2.4 O valor da expressão `X(4)` é 13:

`n`	`X(n)`
1	1
2	2
3	$2 + 3 \times 1 = 5$
4	$5 + 4 \times 2 = 13$

2.2.5 A função está errada pois nem sempre reduz uma instância grande a uma instância menor. Se `n` vale 1, por exemplo, a expressão `n/3 + 1` também vale 1.

2.3.3 Soma recursiva. A função recebe um vetor `v` e um número `n >= 0` e devolve a soma dos elementos positivos de `v[0..n-1]`.

```
int Soma (int v[], int n) {
   if (n == 0) return 0;
   else {
      int s = Soma (v, n - 1);
      if (v[n-1] > 0) s += v[n-1];
      return s; } }
```

2.3.6 A função abaixo está correta mas é muito ineficiente, pois `F(n)` é recalculado muitas vezes para um mesmo `n`. Por exemplo, para calcular `F(4)`, a função calcula `F(2)` duas vezes.

```
int F (int n) {
   if (n == 0) return 0;
   if (n == 1) return 1;
   return F(n - 1) + F(n - 2); }
```

2.3.9 Solução recursiva:

```
int EuclidesR (int m, int n) {
   if (n == 0) return m;
   return EuclidesR (n, m % n); }
```

3.1.1 No início de cada iteração, ou seja, imediatamente antes de cada comparação de `k` com `0`, `x` *é diferente de todos os elementos de* `v[k+1..n-1]`. Exercício: Prove (por indução) que o invariante está correto. Em seguida, prove que o algoritmo está correto.

3.1.2 Esta variante é tão boa quanto a que está no texto.

3.1.3 Erro: o teste "`if (v[k] == x)`" dá resultados imprevisíveis quando `k` está fora do intervalo `0..n-1`.

3.1.5 Com esta decisão de projeto, faz mais sentido percorrer o vetor do começo para o fim:

```
int Busca (int x, int v[], int n) {
   k = 0;
   while (k < n && v[k] != x) k += 1;
   return k; }
```

Para evitar o grande número de comparações de k com n, podemos postar uma "sentinela" em $v[n]$:

```
   k = 0;
   v[n] = x;   /* sentinela */
   while (v[k] != x) k += 1;
   return k;
```

3.1.6 (As perguntas supõem, implicitamente, que o valor inicial de `j` é `0` e não `1`.) Não faz sentido trocar "`x = v[0]`" por "`x = 0`" pois o valor de um elemento máximo do vetor pode ser negativo. Se trocarmos "`x = v[0]`" por "`x = INT_MIN`", a função continua correta mas fica deselegante, pois `INT_MIN` depende do computador e não da função em si. Se trocarmos "`x < v[j]`" por "`x <= v[j]`", a função fica ligeiramente menos eficiente.

3.2.1 Código correto mas deselegante e muito ineficiente. Deveria invocar a recursão somente depois de "`if (x == v[n-1]) return 1`".

3.3.2 Deselegante: a atribuição "v`[`n`- 1] = 0`" é inútil.

3.3.3 Deselegante: "`if (`k `<` n `- 1)`" é supérfluo.

3.5.1 O código não é mais eficiente que o exercício 3.5.2, mas pelo menos não é tão torto.

3.5.2 Ineficiente e deselegante. A alteração do valor de `i` dentro do `for` controlado por `i` é uma péssima maneira de obter o efeito desejado. (A comparação supérflua "`i < n`" dentro do `if` também é deselegante e supérflua.)

4.4.1 Muito deselegante. O segundo parâmetro é inútil. O "`if`" é supérfluo.

4.4.2 A função `Remv` abaixo recebe o endereço `p` de uma célula qualquer de uma lista não vazia `lst`. Se `p` $\neq$ `NULL`, a função supõe que `p->seg` $\neq$ `NULL` e remove a célula cujo endereço é `p->seg`. Se `p` = `NULL`, a função remove a célula cujo endereço é `lst`. Em ambos os casos, a função devolve o endereço da nova lista.

```
célula *Remv (célula *lst, célula *p) {
   célula *lixo;
   if (p == NULL) {
      lixo = lst; lst = lixo->seg; }
   else {
      lixo = p->seg; p->seg = lixo->seg; }
   free (lixo);
   return lst; }
```

Uma solução mais sofisticada usa um ponteiro–para–ponteiro como parâmetro. A função `Remvpp` abaixo recebe o endereço `pp` de um ponteiro que aponta para uma célula (é claro que `pp` deve ser diferente de `NULL`) e remove a célula cujo endereço é `*pp`.

```
void Remvpp (célula **pp) {
   célula *lixo;
   lixo = *pp;
   *pp = lixo->seg;
   free (lixo); }
```

Para remover a primeira célula de uma lista `lst`, o usuário diz `Remvpp (&lst)`. A função atualiza o valor do ponteiro `lst` de tal modo que ele continue apontando para a (primeira célula da) nova lista. Para remover a sucessora da célula cujo endereço é `p` basta dizer `Remvpp (&p->seg)`. (A expressão "`&p->seg`" é interpretada como "`&(p->seg)`", conforme o Seção J.5.)

4.5.2 Não há como fazer isso.

4.5.3 Insere nova célula com conteúdo `y` em uma lista encadeada (com ou sem cabeça, não importa). O segundo parâmetro da função é um ponteiro–para–ponteiro. Para inserir imediatamente após uma célula cujo endereço é `p`, invoque a função com segundo argumento `&p->seg` (veja Seção J.5). Para inserir antes da primeira célula de uma lista `lst`, invoque a função com segundo argumento `&lst` (o valor de `lst` será atualizado nesse caso).

```
void Insere (int y, célula **pp) {
   célula *nova;
   nova = malloc (sizeof (célula));
   nova->conteúdo = y;
   nova->seg = *pp;
   *pp = nova; }
```

4.7.1 Copia vetor para lista: recebe um vetor `v[0..n-1]` e devolve uma lista encadeada sem cabeça com o mesmo conteúdo do vetor.

```
célula *CopiaVetorParaLista (int v[], int n) {
   célula *lst = NULL;
   int i;
   for (i = n-1; i >= 0; i--) {
      célula *p;
      p = malloc (sizeof (célula));
      p->conteúdo = v[i];
      p->seg = lst;
      lst = p; }
   return lst; }
```

4.7.7 Ponto médio de lista:

```
célula *Meio (célula *p) {
   célula *m;
   m = p;
   while (p != NULL && p->seg != NULL) {
      m = m->seg;
      p = p->seg->seg; }
   return m; }
```

5.2.2 Seja v uma cidade no vetor `f[0..t-1]`. Segue dos invariantes 1 e 2 que $d[v]$ é a distância de o a v. Seja w uma cidade fora de `f[0..t-1]`. Suponha que existe um caminho de o a w. Como $o =$ `f[0]`, o invariante 3 garante que o caminho passaria por alguma cidade em `f[s..t-1]`. Mas isso é impossível, pois `s` $=$ `t`.

6.3.3 Esta variante da função `InfixaParaPosfixa` tira proveito dos recursos sintáticos da linguagem C. A parte inicial do código foi omitida pois é idêntica à do texto.

```
for (j = 0, i = 1; infix[i] != '\0'; i++) {
   switch (infix[i]) {
      case '(': p[t++] = infix[i];
                break;
      case ')': while ((x = p[--t]) != '(') posfix[j++] = x;
                break;
      case '+':
      case '-': while ((x = p[t-1]) != '(') {
                   posfix[j++] = x;
                   --t; }
                p[t++] = infix[i];
```

```
                break;
        case '*':
        case '/': while ((x = p[t-1]) != '(' && x != '+' && x != '-') {
                     posfix[j++] = x;
                     --t; }
                  p[t++] = infix[i];
                  break;
        default:  posfix[j++] = infix[i]; } }
```

7.2.2 No início de cada iteração, imediatamente antes da comparação de j com n, temos $v[j-1] < x$. (Esta relação vale até mesmo no começo da primeira iteração se estivermos dispostos a imaginar que $v[-1]$ vale $-\infty$.)

7.3.1 Correto mas ligeiramente deselegante: não é preciso tratar em separado dos casos "`v[n-1] < x`" e "`x <= v[0]`".

7.3.5 As variáveis e e d são sempre ímpares.

7.5.3 Não há como evitar o problema. Mas é possível adiar o desastre (ou seja, lidar com vetores duas vezes maiores) se trocarmos "`(e+d)/2`" pela expressão equivalente "`e + (d-e)/2`".

7.7.1 Correta (e talvez mais fácil de entender que a função dada no texto) mas ligeiramente deselegante pois os dois "`if`" podem ser eliminados (veja texto).

7.8.3 Versão iterativa da busca binária simplificada:

```
int BuscaBin (int x, int n, int v[]) {
   int e = -1, m, d = n;
   while (e < d - 1) {
      m = (e + d)/2;
      if (v[m] == x) return m;
      if (v[m] < x) e = m;
      else d = m; }
   return -1; }
```

7.8.7 Cálculo de k^n com não mais que $2\lfloor \log_2 n \rfloor$ multiplicações:

```
int Potência (int k, int n) {
   if (n == 0) return 1;
   else {
      int m; int fator;
      m = n/2;
      fator = Potência (k, m);
      if (m + m == n) return fator * fator;
      else return fator * fator * k; } }
```

8.2.4 Correto mas deselegante e um tanto ineficiente.

8.2.7 O problema não tem solução (a menos que os elementos do vetor sejam distintos dois a dois).

8.2.9 Versão recursiva:

```
void InserçãoR (int n, int v[]) {
   if (n > 1) {
      int x, i;
      InserçãoR (n - 1, v);
      x = v[n-1];
      for (i = n-2; i >= 0 && v[i] > x; i--) v[i+1] = v[i];
      v[i+1] = x; } }
```

8.4.4 Suponhamos que as células da lista são do tipo `célula`:

```
struct cel {int valor; struct cel *seg;};
typedef struct cel célula;
```

Nossa lista tem célula-cabeça. A função de ordenação recebe um ponteiro para a célula cabeça e não precisa devolver nada:

```
void OrdenaPorInserção (célula *lst) {
   célula *a, *aa, *c; *cc;
   c = lst;
   while (c->seg != NULL) {
      a = lst;
      while (a->seg->valor < c->seg->valor) a = a->seg;
      if (a == c) c = c->seg;
      else {
         aa = a->seg; cc = c->seg;
         a->seg = cc; c->seg = cc->seg;
         cc->seg = aa; } } }
```

9.1.2 O código fica incorreto. (Dê detalhes.)

9.1.5 Não funciona. Entra em *loop* ou produz *segmentation fault*. (Dê detalhes.)

9.1.6 O consumo de tempo é quadrático. (Dê detalhes.)

10.2.4 Muito ineficiente. (Dê detalhes.)

10.3.3 Não escolhe o maior dos filhos. (Dê um exemplo.)

10.4.2 Não. (Dê um contraexemplo simples.)

11.2.3 Os testes "`if (j < k)`" e "`if (j < r)`" são supérfluos.

11.2.6 No início de cada passagem por `A`, $v[p\,..\,r]$ é uma permutação do vetor original, $i \leq j+1$, $p \leq j \leq r$, $v[p{+}1\,..\,j] \leq c$ e $v[j{+}1\,..\,r] > c$.

11.2.5 Invariantes: no início de cada iteração controlada por `while (1)`, $v[p\,..\,r]$ é uma permutação do vetor original, $i \leq j+1$, $v[p{+}1\,..\,i{-}1] \leq c$ e $v[j{+}1\,..\,r] > c$.

11.3.1 A execução da função pode não parar. (Mostre um exemplo.)

11.3.2 A execução da função pode não parar. (Mostre um exemplo.)

11.3.6 Função `Quicksort` reescrita com `while`:

```
void Quicksrt (int p, int r, int v[]) {
   while (p < r) {
      int j = Separa (p, r, v);
      Quicksrt (p, j - 1, v);
      p = j + 1; } }
```

11.3.7 Resta apenas escrever a documentação correta da função:

```
void Qcksrt (int p, int r, int v[]) {
   int j = Separa (p, r, v);
   if (p < j - 1) Qcksrt (p, j - 1, v);
   if (j + 1 < r) Qcksrt (j + 1, r, v); }
```

11.3.11 Versão iterativa do Quicksort. Usa duas pilhas: `pilhap[0..t]` armazena os extremos inferiores dos subvetores e `pilhar[0..t]` armazena os extremos superiores:

```
pilhap[0] = p; pilhar[0] = r; t = 0;
while (t >= 0) {
   p = pilhap[t]; r = pilhar[t]; t--;
   while (p < r) {
      j = Separa (p, r, v);
      t++; pilhap[t] = p; pilhar[t] = j - 1;
      p = j + 1; } }
```

12.4.3 Enumeração das combinações k a k de $1, 2, \ldots, n$. A função recebe $1 \leq k \leq n$ e imprime, em ordem lexicográfica, todas as subsequências de $1, 2, \ldots, n$ com exatamente k elementos:

```
int *s;
void Combinacoes (int n, int k) {
   s = malloc ((k + 1) * sizeof (int));
   rec (0, 1, k, n); }

void rec (int j, int m, int k, int n) {
   if (j == k) imprima (k);
   else {
      if (m <= n - k + j + 1) {
         s[j+1] = m;
         rec (j + 1, m + 1, k, n);
         rec (j, m + 1, k, n); } } }
```

13.2.1 Palavra `*xxxxx` e texto `xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx`. (Generalize este exemplo.)

13.3.3 Tente a sentinela `b[`n`+1] = b[`n`]`.

14.2.6 A função recebe o endereço `r` de uma árvore binária não vazia e devolve o endereço do primeiro nó na ordem e-r-d:

```
nó *Primeiro (árvore r) {
   while (r->esq != NULL) r = r->esq;
   return r; }
```

15.3.1 A função abaixo recebe o endereço `eer` do endereço de uma árvore de busca e o endereço de um nó avulso `novo`. Insere `novo` no lugar correto da árvore e atualiza `*eer` de modo que esse seja o endereço da nova árvore. (Veja o livro de Masters [12].)

```
void Insere (árvore *eer, nó *novo) {
   nó *p, *r;
   r = *eer; p = NULL;
   while (*eer != NULL) {
      p = *eer;
      if (p->chave > novo->chave) eer = &(p->esq);
      else eer = &(p->dir); }
   *eer = novo;
   if (p != NULL) *eer = r; }
```

15.3.2 Recebe uma árvore de busca `r` e nó avulso `novo` com campos `chave`, `esq` e `dir` preenchidos. Insere `novo` e devolve a raiz da nova árvore de busca.

```
árvore InsereR (árvore r, no *novo) {
   if (r == NULL) return novo;
   else {
      if (r->chave > novo->chave) r->esq = InsereR (r->esq, novo);
      else r->dir = InsereR (r->dir, novo);
      return r; } }
```

A.1.1 Fonte tipográfica de espaçamento variável não é adequada para exibir código de programas. É muito melhor usar fonte de `espaçamento fixo`.

A.3.2 Leiaute corrigido:

```
Em 1959 e nas décadas seguintes nenhum programador Cobol
poderia imaginar que os programas de computador que estava
criando ainda estariam em operação no fim do século. Poucos se
lembram hoje de que os primeiros PCs possuíam apenas 64 Kbytes
de memória. Como a quantidade de memória disponível era
pequena, usavam-se muitos truques para economizar esse recurso.
Para representar o ano, armazenava-se (por exemplo) "85" em vez
de "1985". Com a chegada do ano 2000, essa codificação
econômica transformou-se em um erro em potencial.
```

A.3.4 Leiaute corrigido:

```
esq = 0; dir = N - 1;
i = (esq + dir)/2;   /* índice do "meio" de R[] */
while (esq <= dir && R[i] != X) {
```

```
    if (R[i] < X) esq = i + 1;
    else dir = i - 1;   /* novo índice do "meio" de R[] */
    i = (esq + dir)/2;
}
```

I.1.1 Os números produzidos por `rand` não são verdadeiramente aleatórios. Não se pode garantir que eles sejam aleatoriamente pares e ímpares. Dependendo da implementação, os números gerados por `rand` podem ser alternadamente pares e ímpares.

I.1.2 Dependendo do valor de `RAND_MAX`, expressões da forma `RAND_MAX * i` podem produzir um *overflow* aritmético.

Bibliografia

[1] J.L. Bentley. *More Programming Pearls: Confessions of a Coder*. Addison-Wesley, 1988.

[2] J.L. Bentley. *Programming Pearls*. ACM Press, segunda edição, 2000.

[3] R.S. Boyer and J.S. Moore. A fast string searching algorithm. *Communications of the ACM*, 20:762–772, 1977.

[4] C. Charras and T. Lecroq. Exact String Matching Algorithms. Internet: `http://www-igm.univ-mlv.fr/~lecroq/string/index.html`.

[5] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press and McGraw-Hill, segunda edição, 2001.

[6] E. W. Dijkstra. On the cruelty of really teaching computing science. Manuscript EWD1036, 1988. In: *The E. W. Dijkstra Archive*, `http://www.cs.utexas.edu/users/EWD/index10xx.html`, 1988. Manuscript EWD1036.

[7] D. Gries. *The Science of Programming*. Springer, 1981.

[8] J. Harrison. Sorting Algorithms. Internet: `http://www.cs.ubc.ca/spider/harrison/Java/sorting-demo.html`.

[9] C.A.R. Hoare. Quicksort. *Computer Journal*, 5:10–15, 1962.

[10] D.E. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*. Addison-Wesley, segunda edição, 1973.

[11] D.E. Knuth and S. Levy. *The CWEB System of Structured Documentation*. Addison-Wesley, 1994. Internet: `http://www-cs-staff.stanford.edu/~knuth/cweb.html`.

[12] D. Masters. *C: An Introduction with Advanced Applications*. Prentice Hall, 1991.

[13] P. Morin. Sorting Algorithms. Internet: `http://cg.scs.carleton.ca/~morin/misc/sortalg/`.

[14] *Problem Set Archive.* Internet: `http://online-judge.uva.es/problemset/`.

[15] E.S. Roberts. *The Art and Science of C: a Library-Based Introduction to Computer Science.* Addison-Wesley, 1995.

[16] E.S. Roberts. *Programming Abstractions in C: a Second Course in Computer Science.* Addison-Wesley, 1998.

[17] F. Ruskey. Combinatorial Object Server. Internet: `http://theory.cs.uvic.ca/cos.html`.

[18] R. Sedgewick. *Algorithms in C, Parts 1–4.* Addison-Wesley Longman, third edition, 1998.

[19] S.S. Skiena and M.A. Revilla. *Programming Challenges (The Programming Contest Training Manual).* Springer, 2003. Internet: `http://www.programming-challenges.com/`.

[20] R. Ueda. Dicionário br.ispell. Internet: `http://www.ime.usp.br/~ueda/br.ispell/`.

[21] *Wikipedia.* Internet: `http://en.wikipedia.org/`.

[22] J.W.J. Williams. Algorithm 232 (Heapsort). *Communications of the ACM*, 7:347–348, 1964.

[23] N. Ziviani. *Projeto de Algoritmos (com implementações em Pascal e C).* Thomson, segunda edição, 2004.

Termos técnicos em inglês

address	endereço
array	vetor
binary search	busca binária
binary tree	árvore binária
`break`	interrompa, escape
call by value	invocação por valor
cell	célula
character	caractere
dangling pointer	ponteiro solto
data structure	estrutura de dados
data type	tipo de dados
de-queue	remover da fila
delete	remover, apagar
depth	profundidade
derangement	desarranjo
`do`	faça
`else`	senão
enqueue	inserir na fila
field	campo
file	arquivo
floor	piso
`for`	para
head cell	célula-cabeça (de lista)
header file	arquivo-interface
heap	heap
`if`	se
indentation	indentação
inorder traversal	varredura e-r-d
input	entrada
instance	instância
layout	leiaute
leaf	folha (de árvore)
length	comprimento
linked list	lista encadeada
merge	intercalar

node	nó
output	saída
overflow	transbordamento
performance	desempenho
pointer	ponteiro, apontador
pop (from stack)	desempilhar
postorder traversal	varredura e-d-r
preorder traversal	varredura r-e-d
push (on stack)	empilhar
queue	fila
random	aleatório
randomized	aleatorizado
record	registro, struct
`return`	devolva
root	raiz
scan	varrer, esquadrinhar
search tree	árvore de busca
seed	semente
sort	ordenar
sorting	ordenação
stable sort	ordenação estável
stack	pilha
string	string
string matching	busca de palavra
string searching	busca de palavra
`struct`	registro, struct
top (of stack)	topo (de pilha)
two's-complement	complemento-de-dois
`while`	enquanto
wrapper function	função-embalagem

Índice remissivo